NASA/TP-2007-214608

Incorporation of Half-Cycle Theory Into Ko Aging Theory for Aerostructural Flight-Life Predictions

William L. Ko, Van T. Tran, and Tony Chen
NASA Dryden Flight Research Center
Edwards, California

January 2007

NASA STI Program ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program is operated under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI program, see the following:

Access the NASA STI program home page at http://www.sti.nasa.gov.

- E-mail your question via the Internet to help@sti.nasa.gov.

- Fax your question to the NASA STI Help Desk at (301) 621-0134.

- Phone the NASA STI Help Desk at (301) 621-0390.

- Write to:
NASA STI Help Desk
NASA Center for AeroSpace Information
7121 Standard Drive
Hanover, MD 21076-1320

NASA/TP-2007-214608

Incorporation of Half-Cycle Theory Into Ko Aging Theory for Aerostructural Flight-Life Predictions

William L. Ko, Van T. Tran, and Tony Chen
NASA Dryden Flight Research Center
Edwards, California

National Aeronautics and
Space Administration

Dryden Flight Research Center
Edwards, California 93523-0273

January 2007

Cover art: NASA Dryden Flight Research Center, photograph EC04-0029-17.

NOTICE

Use of trade names or names of manufacturers in this document does not constitute an official endorsement of such products or manufacturers, either expressed or implied, by the National Aeronautics and Space Administration.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

CONTENTS

ABSTRACT

The half-cycle crack growth theory was incorporated into the Ko closed-form aging theory to improve accuracy in the predictions of operational flight life of failure-critical aerostructural components. A new crack growth computer program was written for reading the maximum and minimum loads of each half-cycle from the random loading spectra for crack growth calculations and generation of in-flight crack growth curves. The unified theories were then applied to calculate the number of flights (operational life) permitted for B-52B pylon hooks and Pegasus® adapter pylon hooks to carry the Hyper-X launching vehicle that air launches the X-43 Hyper-X research vehicle. A crack growth curve for each hook was generated for visual observation of the crack growth behavior during the entire air-launching or captive flight. It was found that taxiing and the takeoff run induced a major portion of the total crack growth per flight. The operational life theory presented can be applied to estimate the service life of any failure-critical structural components.

NOMENCLATURE

A crack location parameter (A = 1.12 for a surface or edge crack)

a depth (crack size) of semi-elliptic surface crack, in

a_{i-1} crack size at the end of the (i – 1)-th half cycle, in

a_c^o operational (final) crack size associated with operational load $V_{\max}^o$, in, $= \dfrac{Q}{\pi}\left(\dfrac{K_{IC}}{AM_k f \sigma^*}\right)^2 = \dfrac{a_c^p}{f^2}$

a_c^p proof (initial) crack size associated with proof load V^*, in, $= \dfrac{Q}{\pi}\left(\dfrac{K_{IC}}{AM_K \sigma^*}\right)^2$

a_1 crack size at the end of the first flight, in, $= a_c^p + \Delta a_1$

C coefficient of Walker crack growth equation, $\dfrac{\text{in}}{\text{cycle}}\left(\text{ksi}\sqrt{\text{in}}\right)^{-\text{m}}$

c half length of semi-elliptic surface crack, in

$E(k)$ complete elliptic function of the second kind, $= \int_0^{\pi/2} \sqrt{1 - k^2 \sin^2\phi}\, d\phi$

F_1^* number of operational flights based on the first fight load data

f operational load factor associated with the worst half cycle of random load spectrum, $= \dfrac{V_{max}^o}{V^*} = \dfrac{\sigma_{max}^o}{\sigma^*} = \sqrt{\dfrac{a_c^p}{a_c^o}}$

HXLV Hyper-X launch vehicle

h thickness of hook, in

K_{IC} mode I critical stress intensity factor, ksi $\sqrt{\text{in}}$

$K_{\max}$ mode I stress intensity factor associated with $\sigma_{\max}$, ksi $\sqrt{\text{in}}$

ΔK mode I stress intensity amplitude associated with stress amplitude, ($\sigma_{\max} - \sigma_{\min}$), ksi $\sqrt{\text{in}}$

$(K_{\max})_i$ mode I stress intensity factor associated with $(\sigma_{\max})_i$ of *i*-th half cycle, ksi $\sqrt{\text{in}}$

ΔK_i mode I stress intensity amplitude associated $[(\sigma_{\max})_i - (\sigma_{\min})_i]$ of *i*-th half cycle, ksi $\sqrt{\text{in}}$

i 1, 2, 3, …. , integer associated with the *i*-th half cycle

j 1, 2, 3, …. , integer associated with the *j*-th half cycle, or the *j*-th flight

k modulus of elliptic function, $= \sqrt{1 - \left(\frac{a}{c}\right)^2}$

M_k flaw magnification factor ($M_k = 1$ for a shallow crack)

m Walker stress intensity factor exponent associated with $(K_{\max})^m$

N_1 number of stress cycles generated during the first flight

N partial stress cycles during flight (fraction of N_1)

n Walker stress-ratio exponent associated with $(1 - R)^n$

Q surface flaw and plasticity factor, $= [E(k)]^2 - 0.212\left(\frac{\sigma^*}{\sigma_Y}\right)^2$

R stress ratio associated with constant amplitude load cycle, $= \frac{\sigma_{\min}}{\sigma_{\max}}$

R^o stress (or load) ratio associated with the worst half-cycle, $= \frac{\sigma^o_{\min}}{\sigma^o_{\max}} = \frac{V^o_{\min}}{V^o_{\max}}$

R_i stress ratio associated with the *i*-th half cycle, $= \frac{(\sigma_{\min})_i}{(\sigma_{\max})_i}$

SRB/DTV solid rocket booster drop test vehicle

V_A B-52B pylon front hook load, lb

V_{BL} B-52B pylon left rear hook load. lb

V_{BR} B-52B pylon right rear hook load, lb

V_{PFL} Pegasus pylon front left hook load, lb

V_{PFR} Pegasus pylon front right hook load, lb

V_{PRL} Pegasus pylon rear left hook load, lb

V_{PRR} Pegasus pylon rear right hook load, lb

VA B-52B pylon front hook

VBL B-52B pylon left rear hook

VBR B-52B pylon right rear hook

VPFL Pegasus pylon front left hook

VPFR Pegasus pylon front right hook

VPRL Pegasus pylon rear left hook

VPRR Pegasus pylon rear right hook

V	applied hook load, lb
V^*	proof load for any hook, lb
$V_{\max}^o$	maximum load of the worst cycle of random load spectrum, lb
$V_{\min}^o$	minimum load of the worst cycle of random load spectrum, lb
Δa_1	amount of crack growth induced at the end of the first flight, in
Δa	amount of a partial crack growth at any time step during the flight, in
δa_i	crack growth increment induced by the i-th half cycle, in
η	stress/load coefficient, ksi/lb
σ^*	tangential stress at critical stress point induced by the proof load V^*, ksi, $= \eta V^*$
σ_A	tangential stress at critical stress point of B-52B pylon front hook induced by V_A, ksi
σ_{BL}	tangential stress at critical stress point of B-52B pylon rear left hook induced by V_{BL}, ksi
σ_{BR}	tangential stress at critical stress point of B-52B pylon rear right hook induced by V_{BR}, ksi
σ_{PFL}	tangential stress at critical stress point of Pegasus pylon front left hook induced by V_{PFL}, ksi
σ_{PFR}	tangential stress at critical stress point of Pegasus pylon front right hook induced by V_{PFR}, ksi
σ_{PRL}	tangential stress at critical stress point of Pegasus pylon rear left hook induced by V_{PRL}, ksi
σ_{PRR}	tangential stress at critical stress point of Pegasus pylon rear right hook induced by V_{PRR}, ksi
$\sigma_{\max}^o$	tangential stress at critical stress point associated with operational peak load, $V_{\max}^o$, ksi
σ_U	ultimate tensile stress, ksi
σ_Y	yield stress, ksi
$\sigma_{\max}$	maximum stress of constant amplitude loading cycles, ksi
$\sigma_{\min}$	minimum stress of constant amplitude loading cycles, ksi
σ_t	tangential stress along hook inner boundary, ksi
$(\sigma_t)_{\max}$	maximum value of σ_t at the stress critical point, ksi
τ_U	ultimate shear stress, ksi
ϕ	angular coordinate for semi-elliptic surface crack, rad
θ_c	angular location of critical stress point, deg
$(\,)_i$	quantity associated with the i-th half cycle of random loading spectrum
$(\,)^*$	quantity associated with proof load

INTRODUCTION

The NASA Dryden B-52B (McDonnell Douglas, St. Louis, Missouri) launch airplane has been used to carry various types of flight research vehicles for high-altitude air-launching tests. The test vehicle is mated to the B-52B pylon through one L-shaped front hook and two identical L-shaped rear hooks. The L-shaped structural geometry will always induce tensile or compressive stress concentration depending on the loading direction (B-52 hooks can have only tensile stress concentrations). The inner curved boundary point of the hook where the tangential tensile stress reaches a maximum is called a critical stress point of the hook and is the potential fatigue crack initiation site.

During the early stages of the flight tests of the solid rocket booster drop test vehicle (SRB/DTV, 49,000 lb) (1983), the two old rear hooks (fabricated with 4340 steel) failed almost simultaneously during towing of the B-52B airplane carrying the SRB/DTV on a relatively smooth taxiway (low-amplitude dynamic loading). A microsurface crack at the critical stress point of each hook escaped detection because of surface masking by plating films. Those fatigue cracks could have been initiated from the past long period of flight test load cycling and the surface corrosion. If the hook failures had occurred during a takeoff run or during captive flight, a catastrophic accident might have occurred. This type of potential accident underscored the need for reliable and accurate calculations of the fatigue crack growths, which could thereby estimate the safe operational flight life of the hooks for each new flight test program.

Recently, the B-52B airplane has been used to carry the Hyper-X launching vehicle that air-launches the X-43 hypersonic flight research vehicle for Mach 7–10 flight tests. The B-52B pylon hooks were intended to carry the total store weight of 40,000 lb (slightly lighter than the SRB/DTV weight 49,000 lb).

The safety of flight tests using B-52B pylon hooks to carry any drop-test vehicle [for example, the Hyper-X launching vehicle (HXLV)] hinges upon the structural integrity of the failure-critical structural components like B-52B pylon hooks and Pegasus® (Orbital Sciences Corporation, Dulles, Virginia) pylon hooks. It is, therefore, of vital importance to accurately determine the safe operational flight life for each of those failure-critical aerostructural components.

Earlier, Ko (refs. 1–6) developed several aging theories for predicting the operational flight life of airborne failure-critical structural components. The most accurate aging theory developed to date was the Ko closed-form aging theory (refs. 5, 6). In this report, the half-cycle crack growth theory will be incorporated into the Ko closed-form aging theory (refs. 5, 6) to improve the accuracy of operational life predictions of failure-critical airborne structural components. A special half-cycle crack growth computer program was written to calculate the crack growth needed for operational life predictions. The enhanced Ko closed-form aging theory was then applied to calculate the number of safe flights permitted for B-52B pylon hooks and Pegasus adapter pylon hooks to carry the HXLV for air-launching the X-43 hypersonic flight research vehicle.

The operational life theory presented in this report can also be applied to estimate the service life of any failure-critical structural components.

THE B-52B AIRPLANE CARRYING THE HYPER-X LAUNCH VEHICLE

Figure 1 shows the B-52B aircraft carrying the HXLV with the X-43 hypersonic flight research vehicle mated to its nose for air-launching flight tests at Mach 7–10. Because the Pegasus booster rocket has a delta wing which prevents the cylindrical booster body to nest closely under the B-52B pylon concave belly, a special adapter called the Pegasus adapter pylon (weighing 2,300 lb) is used to link the B-52B pylon hooks to the HXLV (weighing 37,700 lb). The Hyper-X launch vehicle is carried by the four identical Pegasus adapter pylon hooks, and the Pegasus adapter pylon is, in turn, carried by the B-52B pylon hooks using a double-shear pin to link to the front hook and through the Pegasus pylon adapter-shackles to connect to the two rear hooks of the B-52B pylon. The total weight then carried by the B-52B pylon hooks is 40,000 lb.

The double-shear pin is not fatigue-critical because there is no stress concentration problem. The two Pegasus pylon adapter shackles, however, are highly failure-critical because each shackle contains a rectangular hole with four, sharp, rounded corners in the upper part, and a circular hole in the lower part (ref. 8). Other failure-critical structural components identified are: the L-shaped B-52B front and two rear hooks and the four, identical L-shaped, Pegasus adapter pylon hooks (ref. 8).

The operational flight-life of all the pylon hooks will be analyzed because the actual loading spectra for those components are now available for the application of the half-cycle crack growth theory. The un-instrumented Pegasus adapter shackles were not analyzed because the actual loading spectra do not exist.

THE KO CLOSED-FORM AGING THEORY

The following section will describe the Ko closed-form aging theory. In the formulation of the Ko closed-form aging theory for aerostructural operational life predictions, the following steps are used.

Failure-Critical Structural Components

A complex structure usually contains a certain number of failure-critical structural components, each of which contains a critical stress point. The critical stress point is a boundary point of the structural component where the tangential tensile stress concentration reaches a maximum, and is the potential fatigue crack initiation site. The operational life of the complex structure is then determined by the operational life of the worst failure-critical structural component having the shortest fatigue life (that is, the fastest crack growth rate at the critical stress point). Therefore, in the operational life analysis, the failure-critical structural components must be identified and their stress fields established.

Stress/Load Equation

In the actual flight tests, the strain gages are usually installed in the vicinity of the critical stress point, and are calibrated to record the applied load (such as hook load). After the failure-critical structural components are identified, stress analysis must be performed for each critical structural component to establish the functional relationship between the applied load and the induced tangential stress at the critical stress point (refs. 7–9). For example, if V^* is the proof load, and if σ^* is the induced proof stress at the stress critical point, then the proof stress, σ^*, may be related to the proof load, V^*, through the following stress/load functional relationship in equation (1)

$$\sigma^* = \eta V^* \tag{1}$$

where η is defined as the stress/load coefficient, and is determined from the finite-element stress analysis of the critical structural component (refs. 7–9).

Operational Load Factor

The next information needed in the operational life analysis is the operational load factor, *f* (<1), defined in equation (2) as

$$f = \frac{\sigma^o_{max}}{\sigma^*} = \frac{V^o_{max}}{V^*} < 1 \tag{2}$$

where σ^o_{max} is the operational maximum stress at the critical stress point induced by the operational maximum load, V^o_{max}, of the worst half-cycle of the random loading spectrum. The worst half-cycle is defined as the half-cycle with the maximum stress (load) amplitude, associated with the minimum stress ratio or load ratio as shown in equation (3)

$$(\sigma^o_{\text{max}} - \sigma^o_{\text{min}}) = \sigma^o_{\text{max}}(1 - R^o) = \text{Maximum} \quad ; \quad R^o = \frac{\sigma^o_{\text{min}}}{\sigma^o_{\text{max}}} = \frac{V^o_{\text{min}}}{V^o_{\text{max}}} = \text{ Minimum} \tag{3}$$

where R^o is the stress (or load) ratio associated with the worst half-cycle. The worst half-cycle is to be searched out in light of condition (3) by means of a special load-factor-searching computer code embedded in the newly written crack growth computer program discussed in Appendix B. Keep in mind that the value of V^o_{max} may not necessarily be the peak load of the entire flight-loading spectrum. Past flight load data showed that the operational maximum load, V^o_{max}, usually occurred during the takeoff run because the ground effect induced the maximum crack growth rate.

Crack Size Determinations

In developing the Ko aging theory (refs. 5, 6), the proof (initial) and operational (final) crack sizes $\{a_c^p, a_c^o\}$ at the critical stress point of the failure-critical structural component must be established first. The two crack sizes $\{a_c^p, a_c^o\}$ are associated respectively with the proof and operational stresses $\{\sigma^*, \sigma_{\max}^o\}$ [or proof and operational peak loads $\{V^*, V_{\max}^o\}$], and are to be calculated from crack tip equations (4) and (5) based on the fracture mechanics (refs 1–4).

$$a_c^p = \frac{Q}{\pi}\left(\frac{K_{IC}}{AM_k\sigma^*}\right)^2 = \frac{Q}{\pi}\left(\frac{K_{IC}}{AM_k\eta V^*}\right)^2 \tag{4}$$

$$a_c^o = \frac{Q}{\pi}\left(\frac{K_{IC}}{AM_k\sigma_{max}^o}\right)^2 = \frac{Q}{\pi}\left(\frac{K_{IC}}{AM_k f\eta V^*}\right)^2 = \frac{a_c^p}{f^2} \tag{5}$$

In equations (4) and (5), K_{IC} is the mode I critical stress intensity factor (material dependent), *A* is the crack location parameter (for a surface crack, *A* = 1.12, refs. 1–4), M_k is the flaw magnification factor (for a shallow surface crack, M_k = 1, refs. 1–4), and finally, *Q* is the surface flaw shape and plasticity factor. For an elliptic surface crack (surface length 2*c*, depth *a*), *Q* may be expressed as in equation (6) (refs. 1–4):

$$Q = [E(k)]^2 - 0.212\left(\frac{\sigma^*}{\sigma_Y}\right)^2 \tag{6}$$

In equation (6), σ_Y is the yield stress, and *E*(*k*) is the complete elliptic function of the second kind defined in equation (7)

$$E(k) = \int_0^{\pi/2}\sqrt{1-k^2\sin^2\phi}\,d\phi \tag{7}$$

where ϕ is the angular coordinate for a semi-elliptic surface crack, seen in fig. 2 (refs. 1–4), and *k* is the modulus of the elliptic function defined in equation (8)

$$k = \sqrt{1-\left(\frac{a}{c}\right)^2} \tag{8}$$

Table 1 lists the input data for finding the values of *E*(*k*) from the complete elliptic integral table (ref. 10) for different crack aspect ratios *a*/2*c*. The values of *Q* were then calculated from equation (6). Table 1 lists only typical values of *Q* calculated for the worst stress ratio $\sigma^*/\sigma_Y = 1$.

Table 1. Key data for the calculations of *Q*, equation (6); $\sigma^{*}/\sigma_{Y} = 1$.

$a/2c$	a/c	$k = \sqrt{1-(a/c)^2}$	$sin^{-1}\,k$, deg.	$E(k)$*	Q
0.1	0.2	0.979796	78.463041	1.0506	0.8918
0.2	0.4	0.916515	66.421822	1.1584	1.1299
0.25	0.5	0.866025	60.0	1.2111	1.2548
0.3	0.6	0.8	53.130102	1.2764	1.4172
0.4	0.8	0.6	36.869898	1.4181	1.7990
0.5	1.0	0.0	0.0	$\pi/2$	2.2554

* Obtained from the complete elliptic integral table (ref. 10).

Figure 2 shows the value of *Q* plotted as a function of crack aspect ratio $a/2c$ with stress ratio σ^{*}/σ_{Y} as a parameter. Remember that the values $\{a/2c = 0.25,\ a/2c = 0.5\}$ listed in table 1 are respectively the aspect ratios of the actual initial surface cracks of the failed B-52B pylon old rear left and right hooks (ref. 7).

The Ko Operational Life Equation

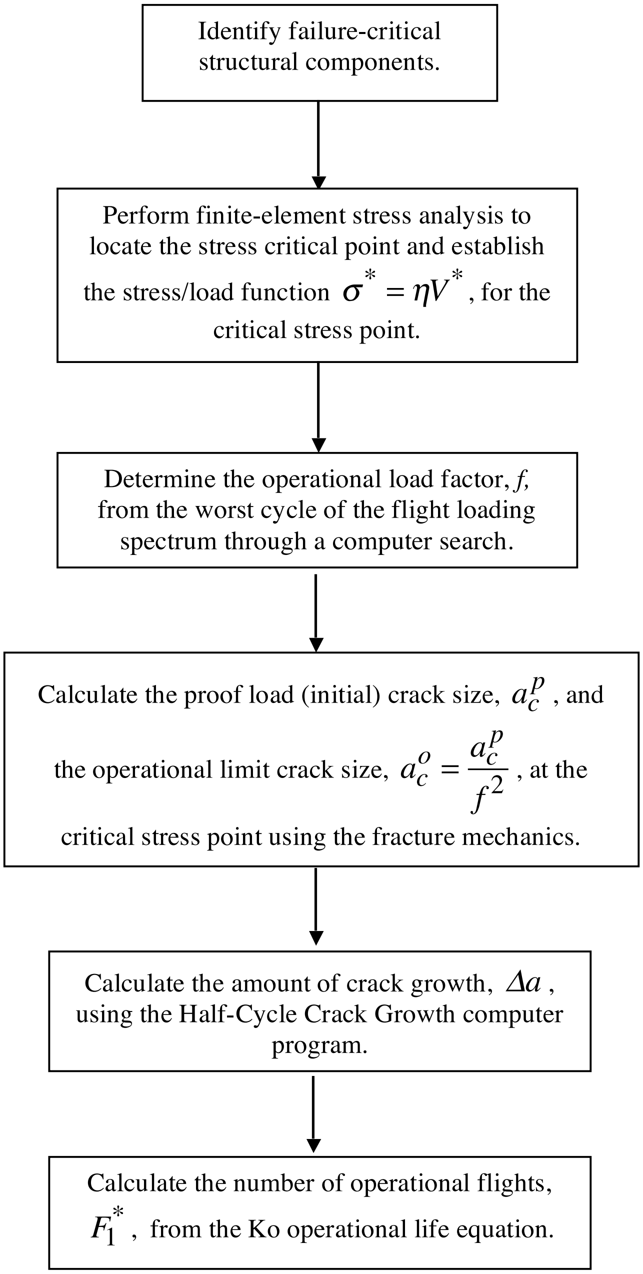
This section describes the basics of the Ko closed-form operational life equations (refs. 5, 6). In the formulation of Ko operational life equations, it was assumed that all the flights last for the same duration of time and induce identical random loading spectra. By representing the random loading spectra with the equivalent-constant-amplitude loading spectra so that the Walker crack growth equation (refs. 3, 4) may be applied, Ko (refs. 5, 6) formulated the closed-form operational life equation (as seen in equation (9) and derived in Appendix A) for the calculations of the number of flights, F_1^*, permitted for each failure-critical aerostructural component.

$$F_1^* = \frac{(a_c^p)^{1-\frac{m}{2}} - (a_c^o)^{1-\frac{m}{2}}}{(a_c^p)^{1-\frac{m}{2}} - (a_1)^{1-\frac{m}{2}}} = \frac{1 - f^{m-2}}{1 - \left(1 + \dfrac{\Delta a_1}{a_c^p}\right)^{1-\frac{m}{2}}} \quad ; \quad a_1 = a_c^p + \Delta a_1 \tag{9}$$

In equation (9), $a_1 (= a_c^p + \Delta a_1)$ is the crack size at the end of the first flight, and Δa_1 is the amount of crack growth induced by the first flight.

In equation (9), the known quantities are: the Walker stress-intensity-factor exponent *m* (refs. 3, 4), the load factor *f* [determined from equation (2)], and the proof (initial) and operational (final) crack sizes $\{a_c^p, a_c^o\}$ [calculated respectively from equations {(4), (5)}]. The only unknown is the crack growth, Δa_1, induced by the first flight. Therefore, the accuracy of the predicted operational flight life, F_1^*, from equation (9) is hinged upon the method of calculations used in determining the crack growth, Δa_1. The step-by-step processes required to use the Ko operational life equation (9) are shown in the following flow chart.

The Ko Operational Life Theory Flow Chart



Identify failure-critical structural components.

Perform finite-element stress analysis to locate the stress critical point and establish the stress/load function $\sigma^* = \eta V^*$, for the critical stress point.

Determine the operational load factor, *f,* from the worst cycle of the flight loading spectrum through a computer search.

Calculate the proof load (initial) crack size, a_c^p, and the operational limit crack size, $a_c^o = \dfrac{a_c^p}{f^2}$, at the critical stress point using the fracture mechanics.

Calculate the amount of crack growth, Δa, using the Half-Cycle Crack Growth computer program.

Calculate the number of operational flights, F_1^*, from the Ko operational life equation.

HALF-CYCLE CRACK GROWTH THEORY

In the calculations of fatigue crack growth under random loading, there are several existing methods (ref. 11). For example,

1) Peak count method
2) Mean crossing peak count method
3) Range count method
4) Range-mean count method
5) Range pair count method
6) Level-crossing count method, and
7) Half-cycle method, etc. (ref. 11).

After reviewing the basics of those different theories, the half-cycle theory was chosen for the present crack growth calculations. The reason being that the half-cycle theory accounts every half-cycle of the random load spectrum without missing any secondary, small-amplitude half-cycles which do not even cross over the mean stress line (ref. 2). The second reason is that the predictions of fatigue life from the half-cycle theory compare fairly well with some existing experimental fatigue data (ref.11, pg. 211, ref. 12).

The half-cycle theory assumes that the amount of crack growth induced by each half-cycle of the random loading spectrum is considered as one-half of a complete cycle of a constant amplitude load spectrum with the same load amplitude. Figure 3 shows the resolutions of the random stress cycles into a series of half-cycles with different loading amplitudes (ref. 2). Under such assumption, the Walker crack growth equation may be used to calculate the incremental crack growth induced by each half-cycle with particular load amplitude.

The Walker Crack Growth Equation

The Walker crack-growth equation for the constant amplitude load spectrum is given in equation (10) by

$$\frac{da}{dN} = C(K_{max})^{m}(1-R)^{n} = C(\Delta K)^{m}(1-R)^{n-m} \tag{10}$$

where C, m, n are material constants. The mode I stress intensity factor, $K_{\max}$, mode I stress intensity amplitude, ΔK, and the stress ratio, R are defined in equations (11), (12), and (13).

$$K_{\max} = AM_k \sigma_{\max} \sqrt{\frac{\pi a}{Q}} \tag{11}$$

$$\Delta K = AM_k (\sigma_{\max} - \sigma_{\min}) \sqrt{\frac{\pi a}{Q}} \tag{12}$$

$$R = \frac{\sigma_{\min}}{\sigma_{\max}} \tag{13}$$

where $\{\sigma_{\max},\ \sigma_{\min}\}$ are respectively the maximum and minimum stresses of the constant amplitude load spectrum. Equation (10) will now be modified to describe the half-cycle crack growth.

The Half-Cycle Crack Growth Equation

In applying the half-cycle theory to calculate the crack growth induced by the random loading spectrum, it is assumed that the incremental amount of crack growth caused by each half-cycle with a particular load amplitude may be considered as a half-cycle of the constant amplitude loading spectrum with the same load amplitude. Therefore, the Walker crack growth equation, (10), may be used to calculate the incremental crack growth induced by each half-cycle of different load amplitude.

If the crack growth increment, *da* in equation (10), is set equal to the crack growth increment, δa_i, induced by the *i*-th (*i* = 1, 2, 3, ….) half cycle (i.e., *da* = δa_i), and the corresponding number of stress cycle increment, *dN,* is set equal to one half cycle (i.e., *dN* = 1/2), then the Walker crack-growth equation (10) becomes the half-cycle crack growth equation for the calculations of half-cycle crack growth increment, δa_i. This half-cycle crack growth is expressed in equation (14).

$$\delta a_i = \frac{C}{2}\left[(K_{max})_i\right]^m (1 - R_i)^n = \frac{C}{2}(\Delta K_i)^m (1 - R_i)^{n-m} \tag{14}$$

where $\{(K_{\max})_i,\ \Delta K_i,\ R_i\}$ are respectively the values of $\{K_{\max},\ \Delta K,\ R\}$ [See equations (11)–(13)} associated with the *i*-th half-cycle given by equations (15), (16), and (17).

$$(K_{\max})_i = AM_k (\sigma_{\max})_i \sqrt{\frac{\pi a_{i-1}}{Q}} \quad ; \quad a_{1-1} = a_0 = a_c^p \ \text{ when } i{=}1 \tag{15}$$

$$\Delta K_i = AM_k\left[(\sigma_{\max})_i - (\sigma_{\min})_i\right]\sqrt{\frac{\pi a_{i-1}}{Q}} \tag{16}$$

$$R_i = \frac{(\sigma_{\min})_i}{(\sigma_{\max})_i} \tag{17}$$

where the subscript i (= 1, 2, 3, ….) is associated with the i-th half-cycle, and a_{i-1} is the cumulated crack size up to the (i –1)-th half-cycle. When i = 1, the crack size a_{i-1} becomes $a_{i-1} = a_{1-1} = a_0 = a_c^p$.

If N is any number of load cycles less than the total load cycles, N_1, induced by the first flight, then the amount of the partial crack growth, Δa , induced by the N load cycles may be obtained from the crack growth equation (18) by summing up all the previous half-cycle crack growth increments, δa_i , up to 2N (not N) cycles as

$$\Delta a = \sum_{i=1}^{2N} \delta a_i \quad ; \quad \left(N \le N_1\right) \tag{18}$$

The summation process of the half-cycle crack growth according to equation (18) is graphically illustrated in fig. 4 (refs. 2–4). Equation (18) is used to calculate the increasing partial crack growths, Δa , with increasing numbers of cycles, $N,$ (or flight time steps) for generating the data set for plotting the crack growth curve (crack growth as a function of flight time) for the critical structural component. When N reached the total number of cycles, N_1 , ($N = N_1$), equation (19) will give the total amount of crack growth, Δa_1 , induced by the first flight. Namely,

$$(\Delta a)_{N=N_1} = \Delta a_1 = \sum_{i=1}^{2N_1} \delta a_i \tag{19}$$

The value of Δa_1 , calculated from equation (19) is to be used as input to equation (9) for the calculation of the number of operational flights F_1^* of the failure-critical structural component.

CRACK GROWTH COMPUTER PROGRAM

To carry out the summation of the half-cycle crack growth increment, δa_i , on the right-hand side of equation (18) or (19), a special crack growth computer program was written. (see Appendix B for details). To use this program and its results, it is necessary to perform the following steps.

1) Create a new data file containing only the required data from the time taxiing begins to the time of test vehicle drop (or the time of complete stop after captive touchdown)

because a flight-test load data file is normally very big covering the ground-sitting portion. Keep in mind that the flight data is the load spectrum and not the stress cycles at the critical stress point.

2) Use a spike remove program to remove noises (spikes) from a flight load spectrum since spikes can add in erroneously big crack growth.

3) Run the crack growth program. This program prompts for an unc3 format input filename. Then, it prompts for some important InterRange Instrumentation Group B (IRIGB) times in milliseconds of start taxiing, takeoff run, cruise power, and drop (or captive stop). After getting all required input data, the crack growth computer program performs the following key functions.

    a. Read the input flight load spectrum. For each channel (associated with each hook) in the input file, the program picks up the maximum and minimum loads for the *i*-th half-cycle, $\left\{(V_{\max})_i, (V_{\min})_i\right\}$. The half-cycle maximum load, $(V_{\max})_i$, is determined when the load is bigger than the two adjacent loads; and conversely, the minimum load, $(V_{\min})_i$, of the same half-cycle is determined when load is smaller than the two adjacent loads.

    b. The loads $\left\{(V_{\max})_i, (V_{\min})_i\right\}$ and their corresponding IRIGB times are saved in asc2 format output files. The names for the asc2 files are simple. For channel vap, the filename is sigma_vap.asc2.

    c. The loads $\left\{(V_{\max})_i, (V_{\min})_i\right\}$ are then converted into the corresponding maximum and minimum stresses, $\{(\sigma_{\max})_i, (\sigma_{\min})_i\}$, of the *i*-th half cycle using equation (1).

    d. Calculate the half-cycle crack growth increment, δa_i, using equation (14), and summing up δa_i over different numbers of load cycles, *N* (or a time step), to generate a data set of different partial crack growths, Δa, from equation (18).

    e. Compute the total crack growth, Δa_1, from equation (19) for the entire flight (from the time of start taxiing to the time of drop or captive stop) for approximately every minute. The times in minutes (zero at start taxiing time) and its corresponding Δa_1 are saved in an unc3 output file.

    f. Determine the worst half-cycle from the loading spectrum during takeoff run and cruise power using the criterion of equation (3) and obtain the operational maximum load, $V_{\max}^{o}$, of the worst half-cycle.

    g. Compute the load factor, *f,* from equation (2).

    h. Calculate the number of operational flights, F_1^* from equation (9) based on the first flight load data.

i. Generate a summary report in text format. This file contains the name of each B-52B hook in the input file, its total crack growth for the first flight, Δa_1, its number of operational flights, F_1^*, its operational load factor *f*, its worst half cycle maximum load $V_{\max}^o$, its worst half-cycle minimum load, $V_{\min}^o$, and the corresponding IRIGB time. It also has the values of the numerator and the denominator that are used to calculate F_1^*.

j. Print on screen the names of the crack growth output file and the summary file.

4) Convert the crack growth file to asc2 format and then to Mircosoft (Redmond, Washington) Excel format.

5) Graphically plot Δa as a function of flight time in minutes using Excel.

OPERATIONAL LIFE ANALYSIS

The Ko aging theory with the half-cycle crack growth theory incorporated, will now be applied to calculate the operational life spans of the three B-52B pylon hooks, and the four Pegasus adapter pylon hooks carrying the HXLV.

Two types of flights were analyzed: 1) air-launching flight, 2) captive flight. The air-launching flight lasted for 106 minutes, counted from the time of B-52B break release for taxiing until the time of air launching (dropping of the HXLV). The captive flight (no air-launching of the HXLV) lasted for 191 minutes, counted from the time of B-52B break release for taxiing and takeoff until the time of complete stop after captive landing.

The purpose of the analysis is to compare the crack growths, Δa_1, induced by the first air-launching and first captive flight, and find out how many air-launching flights will be consumed by each captive flight. The actual flight loading data were used for the operational life calculations.

The B-52B and Pegasus Pylon Hooks

Figures 5–10, taken from reference 8, respectively show the geometry of B-52B pylon hooks (figs. 5, 7), and a typical Pegasus adapter pylon hook (fig. 9). The tangential tensile stress distribution over the inner boundary of each hook, obtained from finite-element stress analysis (figs. 6, 8, 10), is also shown, together with the locations of the critical stress points and the stress/load relationships indicated. The stress/load coefficients, η, for B-52B pylon hooks and Pegasus adapter pylon hooks established from the finite-element stress analysis are summarized in table 2 (taken from ref. 8).

Table 2. Proof loads, V^*, and stress/load coefficients, η, for B-52B pylon hooks and Pegasus adapter pylon hooks.

Hooks	V^*, lb	η, ksi/lb
VA	36,500	7.3522×10^{-3}
VBL	57,819	5.8442×10^{-3}
VBR	57,819	5.8442×10^{-3}
VPFL	75,000	2.4459×10^{-3}
VPFR	75,000	2.4459×10^{-3}
VPRL	75,000	2.4459×10^{-3}
VPRR	75,000	2.4459×10^{-3}

The stress/load coefficients, η, listed in table 2 are to be input to the crack-growth computer program to convert the loading spectrum of each hook into the stress cycles associated with the critical stress point using equation (1).

Flight Load Spectra

Figures 11–17 respectively show the flight load spectra of the B-52B pylon hooks and the Pegasus adapter pylon hooks carrying the HXLV during the takeoff run of the first air-launching flight. The location of the worst half-cycle and the value of the load factor, *f,* are indicated in each figure. The worst half-cycle was located by means of the crack growth computer program searching over the takeoff run portion of each flight load spectrum, and then finding the value of the operational maximum load, $V^o_{\max}$ $(=\sigma^o_{\max}/\eta)$, of the worst half-cycle with minimum load ratio or stress ratio, R^o expressed in equation (20),

$$R^o = \frac{\sigma^o_{\min}}{\sigma^o_{\max}} = \frac{\eta V^o_{\min}}{\eta V^o_{\max}} = \frac{V^o_{\min}}{V^o_{\max}} = \text{ minimum} \tag{20}$$

The value of $V^o_{\max}$ (or $\sigma^o_{\max}$) was then used to calculate the load factor, *f,* for each hook using equation (2).

Crack Growth Calculations

The material properties of B-52B pylon hooks and Pegasus adapter pylon hooks listed in Appendix C were used for the crack growth calculations. In the present crack growth calculations, the surface crack (A = 1.12) at the critical stress point of each hook was assumed to be a very shallow (M_k =1) semi-elliptic surface crack. Only one aspect ratio, $a/2c$ = 1/4 (Q = 1.2548, table

1) was considered. As mentioned earlier, the value $a/2c = 1/4$ is the aspect ratio of the microsurface crack which caused the failure of a B-52B pylon old rear left hook (ref. 7). The crack-growth computer program was then used to read the values of $\{(V_{\max})_i, (V_{\min})_i\}$ for each half-cycle over the loading spectrum, and converted them into the corresponding stresses $\{(\sigma_{\max})_i, (\sigma_{\min})_i\}$ through equation (1) using the η values given in table 2 to calculate the half-cycle crack growth increment, δa_i, using equation (14). Finally, δa_i are summed up to different desired cycles (or time steps) to obtain partial crack growth, Δa, using equation (18) for generating a data set for plotting the crack growth curve for each hook. This process is graphically illustrated in fig. 3 and 4.

Number of Operational Flights

After the total crack growth, Δa_1, induced by the first flight is calculated from equation (19) with the aid of the crack-growth computer program, the operational life equation (9) was then used to calculate the safe number of operational flights, F_1^*, allowed for the B-52B pylon hooks and Pegasus adapter pylon hooks to carry the HXLV for air-launching and captive flights.

RESULTS

The following sections discuss the results of the operational life analysis of the B-52B pylon hooks and the Pegasus adapter pylon hooks carrying the HXLV. This analysis uses the Ko aging theory and is aided by the half-cycle crack growth calculation method.

Crack Growth Curves

The crack growth curve is a very powerful tool for visually observing the crack growth behavior at the critical stress point of each failure-critical component. The crack growth curve for each hook was generated for the following two types of flights: air-launching flight and captive flight.

Air-Launching Flight

Figures 18–20 respectively show the crack growth curves generated for the three B-52B pylon hooks. Those crack growth curves were calculated from equation (18) with the crack growth summation carried out by the crack-growth computer program using the first air-launching flight data. Notice that the crack growth rate for each hook is quite rapid during taxiing because of ground effect, and became more accelerated (illustrated by a steeper slope on the graph) during the takeoff run as the ground-induced vibrations intensified. Once airborne, the ground effect diminished and, therefore, the crack growth rate slowed down considerably and stayed relatively constant (except for encountering wind gusts) until air-launching. The crack growth curve for the B-52B front hook (VA, fig. 18) exhibits the steepest takeoff-run slope as compared with the B-52B two rear hooks (VBL and VBR, figs. 19, 20). The rapid crack growth of the B-52B front hook during the takeoff run could be attributed in part to the overhanging effect of the X-43, which is at a forward distance from the front hook. For the three B-52B pylon hooks, (VA, VBL, VBR), taxiing and takeoff runs combined induced approximately 65, 51, and 41 percent of the respective total crack growth, Δa_1 ,per flight.

Figures 21–24 respectively show the crack growth curves for the four Pegasus adapter pylon hooks (VPFL, VPFR, VPRL, and VPRR). Those crack growth curves were generated from the crack growth computer program in carrying out the summation in equation (18) using the first air-launching-flight load data. The crack growth behavior of the Pegasus adapter pylon hooks is similar to that of the B-52B hooks, but with lower crack growth rates, especially during cruise flight. For the four Pegasus adapter pylon hooks (VPFL, VPFR, VPRL, and VPRR), the taxiing and takeoff run combined induced approximately 45, 60, 64, and 41 precent of the respective total crack growth, Δa_1 , per flight.

Captive Flight

Figures 25–27 respectively show the crack growth curves generated for the three B-52B pylon hooks (VA, VBL, and VBR) using the first captive-flight data. These crack growth curves were calculated from equation (18) with the crack growth summation carried out by the crack-growth computer program. Notice that, for each B-52B pylon hook, the amounts of crack growth and the crack growth rates (shown by slopes on the graphs) during the takeoff phase and the landing phase are quite similar. During the smooth cruise phase, the B-52B airplane encountered only two minor wind gusts (gust 1 and gust 2). The cruising crack growth rate of the front hook (VA, fig. 25) is much slower than those of the two rear hooks (VBL and VBR, figs. 26, 27). At the end of the cruise, three gusts were encountered by the B-52B airplane. The most severe, gust 5 coinciding with the B-52B maneuver, caused the crack growth rate for each hook to increase rapidly (portrayed by steeper slopes). For these three hooks, the fastest crack growth rates occurred during both the takeoff phase and landing phase because of severe ground effects. For the three B-52 hooks (VA, VBL, VBR), the takeoff phase and the landing phase combined contributed approximately 67, 54, and 51 percent of the respective total crack growth, Δa_1 , per flight. The crack growth rate of the outboard right rear hook (VBR) during cruising flight is slightly faster than that of the inboard left rear hook (VBL). This phenomenon was also observed in the air-launching flight-test case (figs. 19, 20).

Figures 28–31 respectively show the crack growth curves generated for the Pegasus adapter pylon hooks (VPFL, VPFR, VPRL, and VPRR) by the crack growth computer program. The program carried out the summation of half-cycle crack growths, calculated by equation (18), associated with the first captive-flight load spectra. The crack growth curves of the Pegasus adapter pylon hooks are similar to those of the B-52B hooks, but with lower crack growth rates, especially during cruise flight. For the four Pegasus adapter pylon hooks (VPFL, VPFR, VPRL, and VPRR), the takeoff phase and landing phase combined induced nearly 51, 59, 71, and 51 percent of the respective total crack growth, Δa_1 , per flight.

Number of Operational Flights

The number of possible operational flights for each of the B-52B pylon hooks and of Pegasus adapter pylon hooks (carrying the HXLV) were calculated from the operational life equation, (9). Flight test data was obtained from two types of test flights, air-launching and captive.

Air-Launching Flight

For the air-launching flight, which lasted for 106 minutes, the key input and output data generated for different hooks are listed in table 3 for crack geometry $a/2c = 0.25$ ($Q = 1.2548$).

Table 3. Key data for the B-52B airplane carrying the Hyper-X launch vehicle (total weight: 40,000 lb); 106-min air-launching flight; $a/2c = 0.25$ ($Q = 1.2548$).

Hooks	V^*, lb	$V^o_{\max}$, lb	f	a^p_c, in	Δa_1, in	F^*_1, flights
VA	36,500	18,065	0.4949	0.0691	1.9258×10^{-4}	304
VBL	57,819	23,227	0.4017	0.0429	2.5367×10^{-4}	186†
VBR	57,819	18,906	0.3270	0.0429	2.5734×10^{-4}	203
VPFL	75,000	34,367	0.4582	0.1455	1.6680×10^{-4}	873
VPFR	75,000	34,623	0.4616	0.1455	1.8326×10^{-4}	790
VPRL	75,000	21,179	0.2824	0.1455	1.4053×10^{-4}	1,323††
VPRR	75,000	21,413	0.2855	0.1455	1.5441×10^{-4}	1,200

† Shortest operational life, †† Longest operational life

Table 3 shows that, among the three B-52B pylon hooks, the rear left hook (VBL) has the shortest life (186 flights), and the front hook (VA) has the longest life (304 flights). Although the crack growths for VBL and VBR are quite close, the higher value of *f* for VBL ($f = 0.4017$) caused the operational life of VBL to be shorter than VBR ($f = 0.3270$).

Among the four Pegasus pylon adapter hooks, the front right hook (VPFR) has the shortest life (790 flights), and the rear left hook (VPRL) has the longest life (1323 flights).

Captive Flight

For the captive flight which had a duration of 191 minutes, the resulting key input and output data for different hooks are listed in table 4 for $a/2c = 0.25$ ($Q = 1.2548$).

Table 4. Key data for the B-52B airplane carrying the Hyper-X launch vehicle (total weight: 40,000 lb); 191-min captive flight; $a/2c = 0.25$ (Q = 1.2548).

Hooks	V^*, lb	$V_{\max}^o$, lb	f	a_c^p, in	Δa_1, in	F_1^*, flights
VA	36,500	17,171	0.4704	0.0691	6.7226×10^{-4}	91
VBL	57,819	21,616	0.3739	0.0429	7.4446×10^{-4}	83†
VBR	57,819	17,875	0.3092	0.0429	5.8556×10^{-4}	92
VPFL	75,000	33,482	0.4464	0.1455	2.1151×10^{-4}	477
VPFR	75,000	34,137	0.4552	0.1455	3.3859×10^{-4}	433
VPRL	75,000	22,565	0.3009	0.1455	3.1070×10^{-4}	586††
VPRR	75,000	21,087	0.2812	0.1455	3.3090×10^{-4}	563

† Shortest operational life, †† Longest operational life

Table 4 shows that, like the air-launching flight, the life of the B-52B pylon rear left hook (VBL) at 83 flights is shorter than the identical rear right hook (VBR) at 92 flights because of higher values of $\{\Delta a_1, f\}$. Among the four identical Pegasus pylon hooks, the front right hook has the shortest life (433 flights), and the rear left hook (VPRL) has the longest life (586 flights). Also, note from table 4 that crack growths, Δa_1, induced by the captive flight are approximately 2–3 times larger than Δa_1 induced by the air-launching flight, therefore, the flight life of each hook is reduced.

Table 5 compares the operational flight life of each hook undergoing air-launching flight and captive flight. The ratio $\dfrac{(F_1^*)_{\text{Air-launching}}}{(F_1^*)_{\text{Captive}}}$ will then give the number of air-launching flights consumed by each captive flight.

Table 5. Summary of available number of flights: B-52B carrying Hyper-X launch vehicle (total weight: 40,000 lb); 106-min air-launching flight; 191 min captive flight.

Hook	F_1^*, flights		Number of air-launching flights consumed by each captive flight
	Air-launching (A)	Captive (C)	$\frac{(A)}{(C)}$
VA	304	91	3.34 (≈ 3)
VBL	186†	83†	2.24 (≈ 2)
VBR	203	92	2.21 (≈ 2)
VPFL	873	477	1.83 (≈ 2)
VPFR	790	433	1.82 (≈ 2)
VPRL	1,323††	586††	2.26 (≈ 2)
VPRR	1,200	563	2.13 (≈ 2)

† Shortest operational life, †† Longest operational life

Note from table 5 that each captive flight consumed 2–3 air-launching flights (depending on the type of hooks) because it had a longer flight duration, encountered more air gusts, experienced aircraft maneuvers, and had a landing phase.

CONCLUSIONS

The half-cycle crack growth theory was incorporated into the Ko closed-form aging theory for accurate crack growth calculations, which would thereby improve the accuracy of predictions of operational life of failure-critical aerostructural components. The unified theories were then used to calculate the number of operational flights permitted for B-52B pylon hooks and Pegasus adapter pylon hooks carrying the HXLV. The highlights of the operational life analysis are:

1) A new crack growth computer program was written to remove the noises, to read the maximum and minimum loads of each half-cycle of the random-flight loading spectra, and then to calculate the crack growths based on the half-cycle crack growth theory.

2) The crack growths calculated from the half-cycle crack growth program should be quite accurate because every half-cycle of each random loading spectrum was counted, including those secondary mini-amplitude half-cycles which did not even cross over the mean stress lines.

3) The crack growth curve generated for each hook using the newly written crack growth computer program is a powerful practical tool for visualization of crack growth behavior at the critical point of each hook during all phases of flight.

4) The crack growth rates are most rapid during the takeoff phase (brake release for taxiing

and takeoff run) and landing phase (touchdown and taxiing to stop) because of ground effect, and induced a large percentage of the total crack growth per flight.

5) Once airborne and during cruise, the crack growth rate decreased significantly, and stayed almost constant, except for encountering wind gusts and aircraft maneuvers.

6) For air-launching flight (the B-52B airplane carrying and launching the HXLV), taxiing and takeoff combined induced approximately 41–65 precent of the total crack growth per flight depending on the types of hooks. The B-52B pylon rear left hook (VBL) has the shortest operational life of 186 flights, and the Pegasus pylon adapter rear left hook (VPRL) has the longest operational life of 1323 flights.

7) For captive flight (the B-52B airplane carrying the HXLV), the takeoff phase and the landing phase combined induced approximately 51–71 percent of the total crack growth per flight depending on the types of hooks. The B-52B pylon rear left hook (VBL) has the shortest operational life of 83 flights, and the Pegasus pylon adapter rear left hook (VPRL) has the longest operational life of 586 flights.

8) Each captive flight is equivalent to 2–3 air-launching flights (depending on the type of hooks) because of longer flight time, encountering more wind gusts, intended aircraft maneuvers, and an additional captive landing phase.

Dryden Flight Research Center
National Aeronautics and Space Administration
Edwards, California, October 12, 2006

APPENDIX A
OPERATIONAL LIFE EQUATIONS

The original Ko closed-form operational life equation (refs. 5, 6) has the mathematical form given by equation (A1)

$$F_1^* = \frac{(a_c^p)^{1-\frac{m}{2}} - (a_c^o)^{1-\frac{m}{2}}}{(a_c^p)^{1-\frac{m}{2}} - (a_1)^{1-\frac{m}{2}}} = \frac{1-\left(\dfrac{a_c^o}{a_c^p}\right)^{1-\frac{m}{2}}}{1-\left(\dfrac{a_1}{a_c^p}\right)^{1-\frac{m}{2}}} \tag{A1}$$

From equation (5), the crack-ratio/load-factor relationship is established as seen in equation (A2).

$$\frac{a_c^o}{a_c^p} = \frac{1}{f^2} \tag{A2}$$

As seen in equation (A3) the crack size at the end of the first flight, a_1, may be expressed in terms of the crack growth, Δa_1, for the first flight as

$$a_1 = a_c^p + \Delta a_1 \tag{A3}$$

In light of equations (A2) and (A3), equation (A1) may be written in more compact form in terms of f in equation (A4)

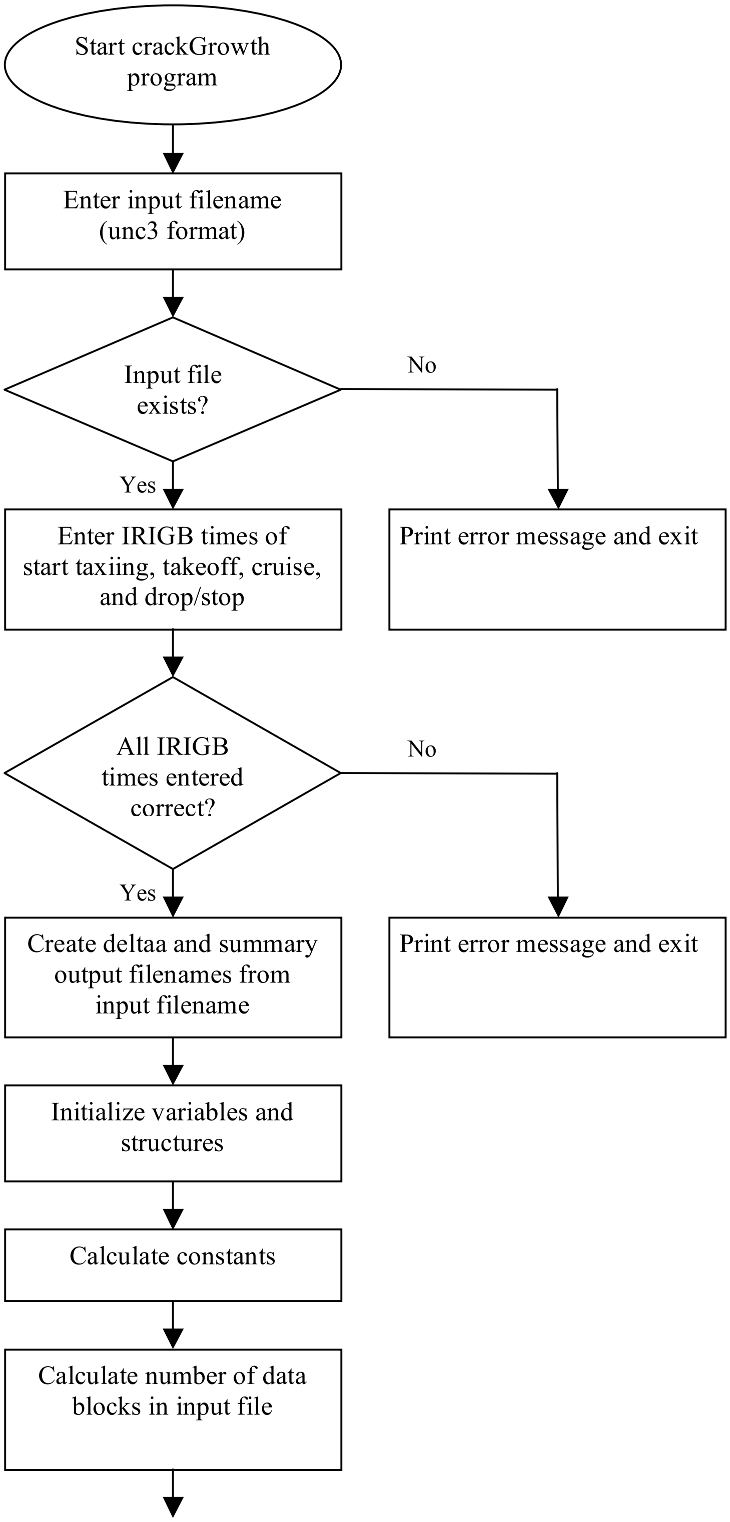
$$F_1^* = \frac{1-\left(\dfrac{1}{f^2}\right)^{1-\frac{m}{2}}}{1-\left(\dfrac{a_c^p + \Delta a_1}{a_c^p}\right)^{1-\frac{m}{2}}} = \frac{1-\left(\dfrac{1}{f}\right)^{2-m}}{1-\left(1+\dfrac{\Delta a_1}{a_c^p}\right)^{1-\frac{m}{2}}} \tag{A4}$$

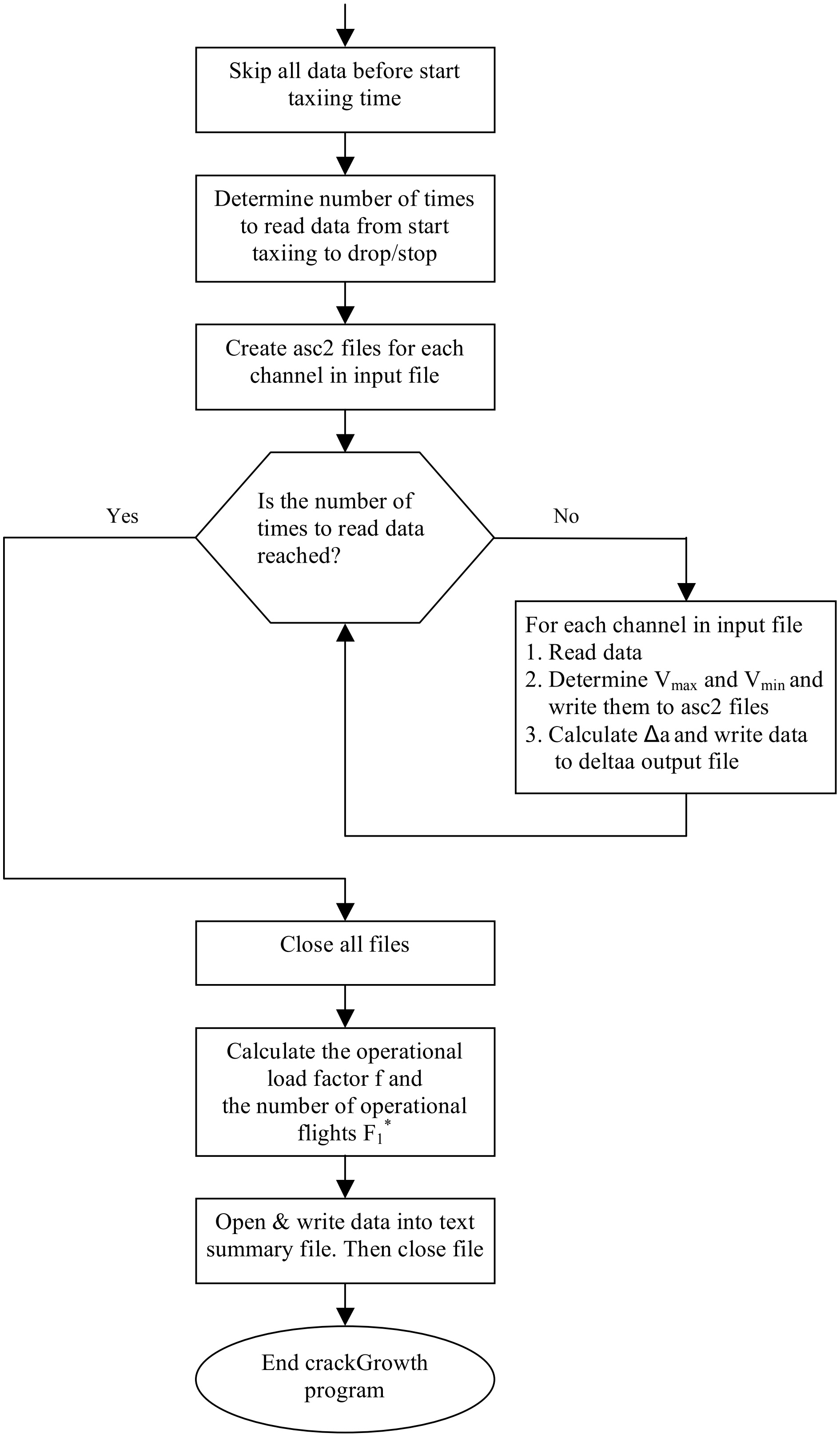
which may be rewritten in equation (A5) as

$$F_1^* = \frac{1 - f^{m-2}}{1 - \left(1 + \frac{\Delta a_1}{a_c^p}\right)^{1-\frac{m}{2}}} \qquad \text{(A5)}$$

which is equation (9), the Ko operational life equation, in the text.

APPENDIX B
CRACK GROWTH COMPUTER PROGRAM


Start crackGrowth program
Enter input filename (unc3 format)
Input file exists?
No
Yes
Enter IRIGB times of start taxiing, takeoff, cruise, and drop/stop
Print error message and exit
All IRIGB times entered correct?
No
Yes
Create deltaa and summary output filenames from input filename
Print error message and exit
Initialize variables and structures
Calculate constants
Calculate number of data blocks in input file


Skip all data before start taxiing time
Determine number of times to read data from start taxiing to drop/stop
Create asc2 files for each channel in input file
Is the number of times to read data reached?
Yes
No
For each channel in input file
1. Read data
2. Determine V_{max} and V_{min} and write them to asc2 files
3. Calculate Δa and write data to deltaa output file
Close all files
Calculate the operational load factor f and the number of operational flights F_1^*
Open & write data into text summary file. Then close file
End crackGrowth program

```
/*****************************************************************************
 * Tittle: crackGrowth.c -- Crack Growth Program                              *
 * Written by: Van T. Tran                                                    *
 * Organization: Aerostructures Branch, RS, NASA Dryden Flight Research Center *
 * Date: August 3, 2004                                                       *
 *                                                                            *
 * Introduction:                                                              *
 *    This program is written in C programming language. It only works with flight test data *
 *    files that have unc3 format. First, it prompts for an input filename. Then, it prompts *
 *    for some important InterRange Instrumentation Group B (IRIGB) times in milliseconds. *
 *    These IRIG times consist of start taxiing, takeoff run, cruise power, and drop/stop. After *
 *    getting all entered inputs, it performs several tasks for each channel in the input file such *
 *    as calculating the accumulated crack growth size, the number of operational flights, *
 *    the operational load factor, creating output files, and generating a summary report. *
 *    When it finishes, it prints on the screen the unc3 format output filename and the text *
 *    summary filename.                                                       *
 *                                                                            *
 *     There are different data formats at Dryden Flight Research Center. The text formats *
 *     consist of asc1 (ASCII 1) and asc2 (ASCII 2). The binary formats include *
 *     unc2 (uncompressed 2), unc3 (uncompressed 3), cmp3 (compressed 3), and *
 *     cmp4 (compressed 4). To convert a file from one format to another format, use *
 *     getdata and/or getdata3 programs.                                      *
 *                                                                            *
 * Inputs:                                                                    *
 *   An unc3 format input file contains loading spectra.                      *
 *                                                                            *
 * Outputs:                                                                   *
 *   1. unc3 output filename = input filename_deltaa.unc3. This file contains IRIGB times *
 *      and the corresponding crack growths, Δa, for all channels in the input file. Δa is *
 *      calculated using the half-cycle crack growth theory.                  *
 *                                                                            *
 *   2. txt output filename = input filename_summary.txt. This file contains the following *
 *      information for all channels in the input file:                      *
 *                                                                            *
 *        - Δa1, the final sum of crack growths                               *
 *                                                                            *
 *        - F1*, the number of operational flights, calculated by using Ko operational *
 *          life equation.                                                    *
 *                                                                            *
 *        - f, the operational load factor.                                   *
 *                                                                            *
 *        - the worst half-cycle V0max, V0min, and the corresponding IRIGB time. *
 *                                                                            *
 *        - the numerator and the denominator used in calculation of F1*.     *
 *                                                                            *
 *   3. asc2 output filename = sigma_channel name.asc2. Each sigma file contains IRIGB *
 *      time, half-cycle $V_{max}$ and $V_{min}$ for each channel in the input file.                 *
 *                                                                                   *
 * Procedure to use crackGrowth program:                                             *
 *                                                                                   *
 *    1. Use getdata3 to convert flight test data file to asc2 format.               *
 *    2. Use vi, xemacs or textedit editor to filter the data so that the input file contains only *
 *       data from start taxiing time to drop/stop time.                             *
 *    3. Use getdata3 to convert the filtered data file to unc3 format input file.   *
 *    4. Use any spike remove program to remove spikes in the input file.            *
 *    5. Run crackGrowth program by type in crackGrowth at the command line.         *
 *    6. Enter all required data as prompted on screen.                              *
 *    7. Program crackGrowth is done when complete message is displayed on screen.   *
 *    8. Use getdata3 program to convert output file xxxx_deltaa.unc3 to asc2 format. *
 *    9. Convert asc2 format to excel format                                         *
 *   10. Use Excel to plot the data                                                  *
 *   11. Use vi, xemacs or textedit editor to read xxxx_summary.txt file.            *
 *   12. Use vi, xemacs or textedit editor to read sigma_xxxx.asc2 files.            *
 *                                                                                   *
 * Initial Release: November 2004                                                    *
 *                                                                                   *
 **********************************************************************************/

 /* Header Files */

 #include <stdio.h>
 #include <string.h>
 #include <stdlib.h>
 #include <fcntl.h>
 #include <unistd.h>
 #include <sys/stat.h>
 #include <sys/ddi.h>
 #include <math.h>

 /* Library subroutines */

 double     sqrt(double x);
 double     pow(double x, double y);

 /* Define constants */

 #define    FALSE                 0
 #define    TRUE                  1
 #define    ERROR                 -1
 #define    NAMES_SIZE            16
 #define    TITLE_LENGTH          80
 #define    FNAME_LENGTH          200
```

```
#define     NUM_BYTES               4
#define     MAXCHANS                30
#define     MAXBUFS                 5000
#define     MAXSIZE                 512
#define     NUM_SECONDS             60

#define     NONE                    0
#define     MIN                     1
#define     MAX                     2
#define     EQUAL                   3

#define     A                       1.12
#define     Mk                      1
#define     Q                       1.2548
#define     PI                      3.14159265359

#define     Eta_FRONT_HOOK          0.0073522
#define     Eta_REAR_HOOK           0.0058442
#define     Eta_PEGASUS             0.0024459

#define     Vstar_FRONT_HOOK        36500.0
#define     Vstar_REAR_HOOK         57819.0
#define     Vstar_PEGASUS           75000.0

#define     Kic_FRONT_HOOK          125.0
#define     Kic_REAR_HOOK           124.0
#define     Kic_PEGASUS             124.0

#define     C_FRONT_HOOK            0.00000000000922
#define     C_REAR_HOOK             0.00000000002944
#define     C_PEGASUS               0.00000000002944

#define     m_FRONT_HOOK            3.6
#define     m_REAR_HOOK             3.24
#define     m_PEGASUS               3.24

#define     n_FRONT_HOOK            2.16
#define     n_REAR_HOOK             1.69
#define     n_PEGASUS               1.69

#define     f_VA                    0.4656
#define     f_VBL                   0.3720
#define     f_VBR                   0.3328
#define     f_VPFL                  0.4585
#define     f_VPFR                  0.4747
#define     f_VPRL                  0.2607
```

```
#define     f_VPRR                          0.2966

/* Define structure */

struct buffer_struct
{
   unsigned long    irig_time;                 /* IRIG B time         */
   float   euc_data[MAXCHANS];          /* EUC data            */
};

/* getdata format record format */

struct
{
   short          size;
   char           text[8];
   char           type[8];
   char           ver[8];
}
format = {sizeof(format), "format  ", "unc 3   ", ".1      "};

/* getdata nChans record format */

struct
{
   short          size;
   char           text[8];
   short          dummy;
   short          count;
}
nchans = {sizeof(nchans), "nChans  ", 0, 0};

/* getdata timekey record format */

struct
{
   short          size;
   char           text[8];
   short          dummy;
   short          count;
}
timekey = {sizeof(timekey), "timekey ", 0, 1000};

/* getdata title record format */

struct
```

```
{
   short        size;
   char         text[8];
   char         titl[TITLE_LENGTH];
}
title = {sizeof(title), "title   "};

/* getdata names record format */

struct
{
   short        size;
   char         text[8];
}
names = {0, "names   "};

/* getdata endhead record format */

struct
{
   short        size;
   char         text[8];
}
endhead = {sizeof(endhead), "endHead "};

/* Define variables */

char     input_file[FNAME_LENGTH];
char     output_file1[FNAME_LENGTH];
char     output_file2[FNAME_LENGTH];

char     f_vap[FNAME_LENGTH];
char     f_vas[FNAME_LENGTH];
char     f_vbrp[FNAME_LENGTH];
char     f_vbrs[FNAME_LENGTH];
char     f_vblp[FNAME_LENGTH];
char     f_vbls[FNAME_LENGTH];
char     f_vprrp[FNAME_LENGTH];
char     f_vprrs[FNAME_LENGTH];
char     f_vprlp[FNAME_LENGTH];
char     f_vprls[FNAME_LENGTH];
char     f_vpfrp[FNAME_LENGTH];
char     f_vpfrs[FNAME_LENGTH];
char     f_vpflp[FNAME_LENGTH];
char     f_vpfls[FNAME_LENGTH];
char      sigma_txt[MAXCHANS][FNAME_LENGTH];
char      chan_name[MAXCHANS][NAMES_SIZE];
char      asc2_name[MAXCHANS][26];

FILE      *fpin;
FILE      *fpout1;
FILE      *fpout2;
FILE      *fp_sigma_txt[MAXCHANS];

struct     buffer_struct  data_buffer;
struct     buffer_struct  data_write1;
struct     buffer_struct  data_write2;
struct     buffer_struct  data_read[MAXBUFS];

double    C_OVER_2[MAXCHANS];
double    deltaa[MAXCHANS][MAXBUFS], sum_a[MAXCHANS],
          sum_deltaa[MAXCHANS];
double    apc[MAXCHANS], apc_FRONT_HOOK, apc_REAR_HOOK, apc_PEGASUS;

float     buff_endfile[MAXSIZE];
float     calc_euc[MAXBUFS], temp_euc[MAXBUFS];
float     stress_coef[MAXCHANS], Kic[MAXCHANS], m[MAXCHANS], n[MAXCHANS];
float     Vstar[MAXCHANS], prev_value[MAXCHANS], end_value[MAXCHANS];
float     min_max_value[MAXCHANS][MAXBUFS], last_data[MAXCHANS];
float     f[MAXCHANS], Vmax[MAXCHANS], Vmin[MAXCHANS], Ri_min[MAXCHANS];
float     numerator[MAXCHANS], denominator[MAXCHANS];

int       num_chans, names_size;
int       buf_size, normal_buf;
int       max_data_read, int_flight[MAXCHANS];

long      unc3_endfile = -1;

short     first_time, last_data_read;
short     prev_type[MAXCHANS], end_type[MAXCHANS], first_type[MAXCHANS];
short     do_calc[MAXCHANS], write_index[MAXCHANS];
short     last_index[MAXCHANS];
short     processed_done[MAXCHANS];

unsigned long   fp, fp_eof, fp_skip, fp_read, data_buff_size, num_data_blocks;
unsigned long   starting_time_entered, starting_time;
unsigned long   drop_time_entered, drop_time, dt;
unsigned long   takeoff_time_entered, takeoff_time;
unsigned long   cruise_time_entered, cruise_time;
unsigned long   time_deltaa;
unsigned long   num_blocks, num_read;
unsigned long   min_max_time[MAXCHANS][MAXBUFS], last_data_time[MAXBUFS],
                Tmax[MAXCHANS];

/* Subroutines used by the main program */

void    get_header_info();
void    write_asc2_header();
void    skip_data();
void    read_data();
void    determine_max_min();
void    calculate_deltaa();
void    calculate_flights();
void    generate_summary();


main(argc, argv)
int argc;
char *argv[];
{
  char      name[NAMES_SIZE];
  int       i, j, k, l;
  double    A_sqre, Mk_sqre, Q_over_PI, Kic_sqre, load_sqre;

  /* Screen display */

  printf("\ncrackGrowth program written by : Van T. Tran");
  printf("\nNASA/Dryden Flight Research Center, Code RS");
  printf("\nInitial Release - November 2004\n\n");

  /* Prompt user for the input filename */

  printf("crackGrowth: Enter input file name ");
  scanf("%s", input_file);

  /* Check if input filename exists */

  if ((fpin = fopen(input_file, "r")) == NULL)
  {
    printf("\nError in openning input file %s\n\n", input_file);
    printf("\ncrackGrowth program terminated\n\n");
    exit (1);
  }

  /* Need start taxiing time in miliseconds */

  printf("\ncrackGrowth: Enter Irigb starting time in msec (integer) ");
```

```
scanf("%d", &starting_time_entered);
printf("Entered Irigb starting time is %d\n\n", starting_time_entered);
starting_time = starting_time_entered * 10;

/* Need takeoff run time in miliseconds */

printf("crackGrowth: Enter Irigb takeoff time in msec (integer) ");
scanf("%d", &takeoff_time_entered);
printf("Entered Irigb takeoff time is %d\n\n", takeoff_time_entered);
takeoff_time = takeoff_time_entered * 10;

/* Need cruise power time in miliseconds */

printf("crackGrowth: Enter Irigb cruise time in msec (integer) ");
scanf("%d", &cruise_time_entered);
printf("Entered Irigb cruise time is %d\n\n", cruise_time_entered);
cruise_time = cruise_time_entered * 10;

/* Need drop or stop time in miliseconds */

printf("crackGrowth: Enter Irigb drop or stop time in msec (integer) ");
scanf("%d", &drop_time_entered);
printf("Entered Irigb drop time is %d\n\n", drop_time_entered);
drop_time = drop_time_entered * 10;

printf("crackGrowth program is running ...Do not interrupt ...\n");

strcpy(output_file1, input_file);
strcpy(output_file2, input_file);

/* Check if the input file has the extension of .unc3 */

if (strstr(input_file, ".unc3") != NULL)
{
  j = strlen(output_file1) - 5;
  strcpy(&output_file1[j], "_deltaa.unc3");
  strcpy(&output_file2[j], "_summary.txt");
}
else
{
  strcat(output_file1, "_deltaa.unc3");
  strcat(output_file2, "_summary.txt");
}

/* Write the output filename */
```

```
if ((fpout1 = fopen(output_file1, "w")) == NULL)
{
  printf("\nError in creating output file %s!", output_file1);
  printf("\ncrackGrowth program terminated\n\n");
  exit (2);
}

if ((fpout2 = fopen(output_file2, "w")) == NULL)
{
  printf("\nError in creating output file %s!", output_file2);
  printf("\ncrackGrowth program terminated\n\n");
  exit (2);
}

/* Initialize variables and structures */

bzero((char *) (&apc), sizeof(apc));
bzero((char *) (&data_read), sizeof(data_read));
bzero((char *) (&data_write1), sizeof(data_write1));
bzero((char *) (&data_write2), sizeof(data_write2));
bzero((char *) (&buff_endfile), sizeof(buff_endfile));
bzero((char *) (&write_index), sizeof(write_index));
bzero((char *) (&stress_coef), sizeof(stress_coef));
bzero((char *) (&deltaa), sizeof(deltaa));
bzero((char *) (&sum_a), sizeof(sum_a));
bzero((char *) (&sum_deltaa), sizeof(sum_deltaa));
bzero((char *) (&do_calc), sizeof(do_calc));
bzero((char *) (&chan_name), sizeof(chan_name));
bzero((char *) (&asc2_name), sizeof(asc2_name));
bzero((char *) (&prev_type), sizeof(prev_type));
bzero((char *) (&Vmax), sizeof(Vmax));
bzero((char *) (&Vmin), sizeof(Vmin));
bzero((char *) (&numerator), sizeof(denominator));

num_chans = 0;
num_blocks = 0;
num_read = 0;
first_time = TRUE;
last_data_read = FALSE;

A_sqre = (double) A*A;
Mk_sqre = (double) Mk*Mk;
Q_over_PI = (double) Q/PI;

Kic_sqre = (double) Kic_FRONT_HOOK*Kic_FRONT_HOOK;
load_sqre = (double) Eta_FRONT_HOOK*Eta_FRONT_HOOK*
```

```
                              Vstar_FRONT_HOOK*Vstar_FRONT_HOOK;
apc_FRONT_HOOK = Q_over_PI * Kic_sqre / (A_sqre * Mk_sqre * load_sqre);

Kic_sqre = (double) Kic_REAR_HOOK*Kic_REAR_HOOK;
load_sqre = (double) Eta_REAR_HOOK*Eta_REAR_HOOK*
                              Vstar_REAR_HOOK*Vstar_REAR_HOOK;
apc_REAR_HOOK = Q_over_PI * Kic_sqre / (A_sqre * Mk_sqre * load_sqre);

Kic_sqre = (double) Kic_PEGASUS*Kic_PEGASUS;
load_sqre = (double) Eta_PEGASUS*Eta_PEGASUS*
                              Vstar_PEGASUS*Vstar_PEGASUS;
apc_PEGASUS = Q_over_PI * Kic_sqre / (A_sqre * Mk_sqre * load_sqre);

/* Move the file position indicator to the end of input file */

fseek(fpin, 0, SEEK_END);

/* Get the pointer at the end of file */

fp_eof = ftell(fpin);
fp_eof -= NUM_BYTES;

/* Move the file position indicator to the beginning of input file */

fseek(fpin, 0, SEEK_SET);
fp = ftell(fpin);

/* Call get_header_info subroutine */

get_header_info();

/* Calculate the number of data blocks in the input file */

num_data_blocks = (fp_eof - fp) / data_buff_size;

/* Skip all data before start taxiing time */

skip_data();

/* Determine the number of times to read data */

max_data_read = (num_data_blocks / normal_buf) + 1;

/* Create an asc2 format file for each channel */

for (i = 0; i < num_chans; i++)
```

```
{
  if (do_calc[i] == TRUE)
  {
   if ((fp_sigma_txt[i] = fopen(sigma_txt[i], "w")) == NULL)
   {
     printf("\nError in creating text file %s!", fp_sigma_txt[i]);
     printf("\ncrackGrowth program terminated\n\n");
     exit (2);
   }
  }
}

/* Create asc2 header for each channel */

write_asc2_header();

/* Read data, determine max & min, and calculate crack growth */

for (i = 0; i < max_data_read; i++)
{
  if (last_data_read == FALSE)
  {
   read_data();
   determine_max_min();
   calculate_deltaa();
  }
}

/* Write end of file to the output file */

fwrite(&unc3_endfile, sizeof(long), 1, fpout1);
fwrite(&buff_endfile, sizeof(float)*MAXSIZE, 1, fpout1);

/* Close all files */

fclose(fpout1);
fclose(fpin);
for (i = 0; i < num_chans; i++)
{
  if (do_calc[i] == TRUE)
  {
   fclose(fp_sigma_txt[i]);
  }
}
printf("\n");
```

```
  /* Calculate the number of operational flights */

  calculate_flights();

  /* Generate a summary text file */

  generate_summary();

  /* Close the summary text file */

  fclose(fpout2);

  /* Print the end message */

  printf("\ncrackGrowth program completed successfully!\n");
  printf("Crack growths are in %s\n", output_file1);
  printf("Summary is in %s\n\n\n", output_file2);
}


/****************************************************************************
 * Subroutine get_header_info()                                             *
 *                                                                          *
 * Description:                                                             *
 *   This subroutine reads the header information in the unc3 input file and write the header  *
 *    information to the unc3 output file. The unc3 output file contains calculated crack       *
 *    growths and times in minutes for all channels. For each channel, all important constants *
 *    are calculated and an asc2 format output file is generated. The asc2 filename has        *
 *    sigma_ following by the channel name and extension asc2. For channel vap, the asc2       *
 *    filename is sigma_vap.asc2.                                           *
 *                                                                          *
 ****************************************************************************/
void get_header_info()
{
  int       i;
  short     str_loc;

  /* Read and write header information */

  fread(&format, sizeof(format), 1, fpin);
  fwrite(&format, sizeof(format), 1, fpout1);

  /* Read the number of input channels and figure out the data buffer size */

  fread(&nchans, sizeof(nchans), 1, fpin);
  fwrite(&nchans, sizeof(nchans), 1, fpout1);
```

```
num_chans = nchans.count;
data_buff_size = num_chans * sizeof(float) + NUM_BYTES;

fread(&timekey, sizeof(timekey), 1, fpin);
fwrite(&timekey, sizeof(timekey), 1, fpout1);

fread(&title, sizeof(title), 1, fpin);
fwrite(&title, sizeof(title), 1, fpout1);

names_size = num_chans * NAMES_SIZE + 10;

/* Read and write channel names */

fread(&names, names_size, 1, fpin);
fwrite(&names, names_size, 1, fpout1);

str_loc = sizeof(names.text);

/* Calculate and determine constants and create sigma filenames */

for (i = 0; i < num_chans; i++)
{
  strncpy(chan_name[i], &names.text[str_loc], NAMES_SIZE);
  str_loc += NAMES_SIZE;
  if (strstr(chan_name[i], "vap") != NULL)
  {
    stress_coef[i] = Eta_FRONT_HOOK;
    Vstar[i] = Vstar_FRONT_HOOK;
    Kic[i] = Kic_FRONT_HOOK;
    apc[i] = apc_FRONT_HOOK;
    C_OVER_2[i] = C_FRONT_HOOK / 2.0;
    m[i] = m_FRONT_HOOK;
    n[i] = n_FRONT_HOOK;
    f[i] = f_VA;
    do_calc[i] = TRUE;
    Ri_min[i] = 1000;
    strcpy(sigma_txt[i], "sigma_vap.asc2");
  }
  else if (strstr(chan_name[i], "vas") != NULL)
  {
    stress_coef[i] = Eta_FRONT_HOOK;
    Vstar[i] = Vstar_FRONT_HOOK;
    Kic[i] = Kic_FRONT_HOOK;
    apc[i] = apc_FRONT_HOOK;
    C_OVER_2[i] = C_FRONT_HOOK / 2.0;
    m[i] = m_FRONT_HOOK;
  n[i] = n_FRONT_HOOK;
  f[i] = f_VA;
  do_calc[i] = TRUE;
  Ri_min[i] = 1000;
  strcpy(sigma_txt[i], "sigma_vas.asc2");
}
else if (strstr(chan_name[i], "vbrp") != NULL)
{
  stress_coef[i] = Eta_REAR_HOOK;
  Vstar[i] = Vstar_REAR_HOOK;
  Kic[i] = Kic_REAR_HOOK;
  apc[i] = apc_REAR_HOOK;
  C_OVER_2[i] = C_REAR_HOOK / 2.0;
  m[i] = m_REAR_HOOK;
  n[i] = n_REAR_HOOK;
  f[i] = f_VBR;
  do_calc[i] = TRUE;
  Ri_min[i] = 1000;
  strcpy(sigma_txt[i], "sigma_vbrp.asc2");
}
else if (strstr(chan_name[i], "vbrs") != NULL)
{
  stress_coef[i] = Eta_REAR_HOOK;
  Vstar[i] = Vstar_REAR_HOOK;
  Kic[i] = Kic_REAR_HOOK;
  apc[i] = apc_REAR_HOOK;
  C_OVER_2[i] = C_REAR_HOOK / 2.0;
  m[i] = m_REAR_HOOK;
  n[i] = n_REAR_HOOK;
  f[i] = f_VBR;
  do_calc[i] = TRUE;
  Ri_min[i] = 1000;
  strcpy(sigma_txt[i], "sigma_vbrs.asc2");
}
else if (strstr(chan_name[i], "vblp") != NULL)
{
  stress_coef[i] = Eta_REAR_HOOK;
  Vstar[i] = Vstar_REAR_HOOK;
  Kic[i] = Kic_REAR_HOOK;
  apc[i] = apc_REAR_HOOK;
  C_OVER_2[i] = C_REAR_HOOK / 2.0;
  m[i] = m_REAR_HOOK;
  n[i] = n_REAR_HOOK;
  f[i] = f_VBL;
  do_calc[i] = TRUE;
  Ri_min[i] = 1000;
  strcpy(sigma_txt[i], "sigma_vblp.asc2");
}
else if (strstr(chan_name[i], "vbls") != NULL)
{
  stress_coef[i] = Eta_REAR_HOOK;
  Vstar[i] = Vstar_REAR_HOOK;
  Kic[i] = Kic_REAR_HOOK;
  apc[i] = apc_REAR_HOOK;
  C_OVER_2[i] = C_REAR_HOOK / 2.0;
  m[i] = m_REAR_HOOK;
  n[i] = n_REAR_HOOK;
  f[i] = f_VBL;
  do_calc[i] = TRUE;
  Ri_min[i] = 1000;
  strcpy(sigma_txt[i], "sigma_vbls.asc2");
}
else if (strstr(chan_name[i], "vprrp") != NULL)
{
  stress_coef[i] = Eta_PEGASUS;
  Vstar[i] = Vstar_PEGASUS;
  Kic[i] = Kic_PEGASUS;
  apc[i] = apc_PEGASUS;
  C_OVER_2[i] = C_PEGASUS / 2.0;
  m[i] = m_PEGASUS;
  n[i] = n_PEGASUS;
  f[i] = f_VPRR;
  do_calc[i] = TRUE;
  Ri_min[i] = 1000;
  strcpy(sigma_txt[i], "sigma_vprrp.asc2");
}
else if (strstr(chan_name[i], "vprrs") != NULL)
{
  stress_coef[i] = Eta_PEGASUS;
  Vstar[i] = Vstar_PEGASUS;
  Kic[i] = Kic_PEGASUS;
  apc[i] = apc_PEGASUS;
  C_OVER_2[i] = C_PEGASUS / 2.0;
  m[i] = m_PEGASUS;
  n[i] = n_PEGASUS;
  f[i] = f_VPRR;
  do_calc[i] = TRUE;
  Ri_min[i] = 1000;
  strcpy(sigma_txt[i], "sigma_vprrs.asc2");
}
else if (strstr(chan_name[i], "vprlp") != NULL)
{
  stress_coef[i] = Eta_PEGASUS;
  Vstar[i] = Vstar_PEGASUS;
  Kic[i] = Kic_PEGASUS;
  apc[i] = apc_PEGASUS;
  C_OVER_2[i] = C_PEGASUS / 2.0;
  m[i] = m_PEGASUS;
  n[i] = n_PEGASUS;
  f[i] = f_VPRL;
  do_calc[i] = TRUE;
  Ri_min[i] = 1000;
  strcpy(sigma_txt[i], "sigma_vprlp.asc2");
}
else if (strstr(chan_name[i], "vprls") != NULL)
{
  stress_coef[i] = Eta_PEGASUS;
  Vstar[i] = Vstar_PEGASUS;
  Kic[i] = Kic_PEGASUS;
  apc[i] = apc_PEGASUS;
  C_OVER_2[i] = C_PEGASUS / 2.0;
  m[i] = m_PEGASUS;
  n[i] = n_PEGASUS;
  f[i] = f_VPRL;
  do_calc[i] = TRUE;
  Ri_min[i] = 1000;
  strcpy(sigma_txt[i], "sigma_vprls.asc2");
}
else if (strstr(chan_name[i], "vpfrp") != NULL)
{
  stress_coef[i] = Eta_PEGASUS;
  Vstar[i] = Vstar_PEGASUS;
  Kic[i] = Kic_PEGASUS;
  apc[i] = apc_PEGASUS;
  C_OVER_2[i] = C_PEGASUS / 2.0;
  m[i] = m_PEGASUS;
  n[i] = n_PEGASUS;
  f[i] = f_VPFR;
  do_calc[i] = TRUE;
  Ri_min[i] = 1000;
  strcpy(sigma_txt[i], "sigma_vpfrp.asc2");
}
else if (strstr(chan_name[i], "vpfrs") != NULL)
{
  stress_coef[i] = Eta_PEGASUS;
  Vstar[i] = Vstar_PEGASUS;
  Kic[i] = Kic_PEGASUS;
  apc[i] = apc_PEGASUS;
```

```
    C_OVER_2[i] = C_PEGASUS / 2.0;
    m[i] = m_PEGASUS;
    n[i] = n_PEGASUS;
    f[i] = f_VPFR;
    do_calc[i] = TRUE;
    Ri_min[i] = 1000;
    strcpy(sigma_txt[i], "sigma_vpfrs.asc2");
  }
  else if (strstr(chan_name[i], "vpflp") != NULL)
  {
    stress_coef[i] = Eta_PEGASUS;
    Vstar[i] = Vstar_PEGASUS;
    Kic[i] = Kic_PEGASUS;
    apc[i] = apc_PEGASUS;
    C_OVER_2[i] = C_PEGASUS / 2.0;
    m[i] = m_PEGASUS;
    n[i] = n_PEGASUS;
    f[i] = f_VPFL;
    do_calc[i] = TRUE;
    Ri_min[i] = 1000;
    strcpy(sigma_txt[i], "sigma_vpflp.asc2");
  }
  else if (strstr(chan_name[i], "vpfls") != NULL)
  {
    stress_coef[i] = Eta_PEGASUS;
    Vstar[i] = Vstar_PEGASUS;
    Kic[i] = Kic_PEGASUS;
    apc[i] = apc_PEGASUS;
    C_OVER_2[i] = C_PEGASUS / 2.0;
    m[i] = m_PEGASUS;
    n[i] = n_PEGASUS;
    f[i] = f_VPFL;
    do_calc[i] = TRUE;
    Ri_min[i] = 1000;
    strcpy(sigma_txt[i], "sigma_vpfls.asc2");
  }
  else
  {
    do_calc[i] = FALSE;
  }
}

/* Read the end header of the input file and write it to the output file */

fread(&endhead, sizeof(endhead), 1, fpin);
fwrite(&endhead, sizeof(endhead), 1, fpout1);
```

```
  /* Get the current value of the file-position pointer */

  fp = ftell(fpin);
}


/**************************************************************************
 * Subroutine write_asc2_header()                                         *
 *                                                                        *
 * Description:                                                           *
 *   This subroutine writes the header information in the asc2 format output files. Each asc2    *
 *   file contains the IRIGB times, maximum loads (Vmax), and minimum loads (Vmin) for    *
 *   each input channel in the input file.                                *
 *                                                                        *
 **************************************************************************/
void write_asc2_header()
{
  int        i, j;
  char       temp_name[] = "             ";

  for (i = 0; i < num_chans; i++)
  {
    bzero((char *) (&temp_name), sizeof(temp_name));
    if (do_calc[i] == TRUE)
    {
      strncpy(temp_name, chan_name[i], 13);
      fprintf(fp_sigma_txt[i], "format  asc 2   .1      \n");
      fprintf(fp_sigma_txt[i], "nChans         1\n");
      fprintf(fp_sigma_txt[i], "names        ");
      fprintf(fp_sigma_txt[i], "%13s", temp_name);
      fprintf(fp_sigma_txt[i], "\n");
      fprintf(fp_sigma_txt[i], "data001 \n");
    }
  }
}


/**************************************************************************
 * Subroutine skip_data()                                                 *
 *                                                                        *
 * Description:                                                           *
 *   This subroutine reads data blocks from the input file until the start taxiing time is reached.  *
 *   It positions the file pointer to the start taxiing time. It also calculates the time interval    *
 *   between two adjacent data points to figure out the number of data blocks in 1 minute.    *
 *                                                                        *
```

```
****************************************************************************/
void skip_data()
{
  int         i, j, k, offset;
  float       delta_t, num_buf;

  /* Check if the start taxiing time is reached */

  for (i = 0; i < num_data_blocks; i++)
  {
    fread(&data_read[i], data_buff_size, 1, fpin);

    if (data_read[i].irig_time >= starting_time)
    {
     /* Read irigb time and data */

     data_write1.irig_time = data_read[i].irig_time;
     fwrite(&data_write1, data_buff_size, 1, fpout1);
     break;
    }
  }

  /* Move the file pointer to the start taxiing time */

  if (i == 0)
  {
    fread(&data_read[1], data_buff_size, 1, fpin);
    offset = -2 * data_buff_size;
  }
  else
  {
    offset = -data_buff_size;
  }

  /* Calculate the time interval between 2 adjacent data points */

  delta_t = (data_read[1].irig_time - data_read[0].irig_time) / 10000.0;

  /* Calculate the number of data blocks containing in 1 minute (60 sec) */

  num_buf = NUM_SECONDS / delta_t;
  normal_buf = (int) num_buf;

  /* Move the file position indicator to the start taxiing time */

  fseek(fpin, offset, SEEK_CUR);
```

```
}

/*****************************************************************************
 * Subroutine read_data()                                                    *
 *                                                                           *
 * Description:                                                              *
 *   This subroutine determines the number of data blocks to read into data_read buffers.   *
 *   After each reading, it checks to see if the drop or stop time is reached. If yes, it sets the   *
 *   last_data_read indicator to TRUE and determines the buffer size for the last data read. If   *
 *   no, it sets the buffer size and move the file pointer to the right position for the next data   *
 *   read.                                                                   *
 *                                                                           *
 *****************************************************************************/
void read_data()
{
  int          i, count, offset;

  num_read++;
  num_blocks = 0;
  bzero((char *) (&data_write1), sizeof(data_write1));

  /* Determine the number of data blocks to read */

  if (first_time == TRUE)
  {
    count = normal_buf + 2;
  }
  else
  {
    count = normal_buf + 1;
  }

  for (i = 0; i < count; i++)
  {
    /* Read data */

    fread(&data_read[i], data_buff_size, 1, fpin);
    num_blocks++;

    /* Check if the drop or stop time is reached */

    if (data_read[i].irig_time >= drop_time)
    {
      /* Indicate the last data read */

      last_data_read = TRUE;
      time_deltaa = data_read[i].irig_time;
      data_write1.irig_time = data_read[i].irig_time;
      buf_size = i+1;
      break;
    }
  }

  /* Check if this is the last data read */

  if (last_data_read == FALSE)
  {
    buf_size = count-1;
    time_deltaa = data_read[buf_size-1].irig_time;
    data_write1.irig_time = data_read[buf_size-1].irig_time;

    /* Move the file pointer to the right position for next data read */

     offset = -data_buff_size;
     fseek(fpin, offset, SEEK_CUR);
  }
}


/****************************************************************************
 * Subroutine determine_max_min()                                           *
 *                                                                          *
 * Description:                                                             *
 *   This subroutine determines Vmax and Vmin loads for the current data read.  *
 *   Vi = Vmax if Vi > Vi-1 and Vi > Vi+1                                   *
 *   Vi = Vmin if Vi < Vi-1 and Vi < Vi+1                                   *
 *   In cases that Vi-1 < Vi < Vi+1 or Vi-1 > Vi > Vi+1, Vi is definitely not Vmax nor Vmin.  *
 *   In these special cases, comparisons will continue beyond i+1 data point. When a Vmax  *
 *   or Vmin is found, its IRIGB time and its value are written into the asc2 file and also  *
 *   stored in two dimensional min_max_value arrays to be used later for calculating  *
 *   crack growths.                                                         *
 *                                                                          *
 ***************************************************************************/
void determine_max_min()
{
  char      ch_name[] = "        ";
  float     save_euc[MAXBUFS];
  float     calc_stress, min_max;
  int       i, j, k, l, index, start_index, cur_index;
  short     max_min_found;

  /* Reset indicators and variables */
```

```
bzero((char *) (&processed_done), sizeof(processed_done));
bzero((char *) (&min_max_value), sizeof(min_max_value));
bzero((char *) (&min_max_time), sizeof(min_max_time));

/* For each channel */

for ( j = 0; j < num_chans; j++)
{
  bzero((char *) (&temp_euc), sizeof(temp_euc));

  /* Load data into working temp_euc buffers */

  if (do_calc[j] == TRUE)
  {
    if (last_data_read == FALSE)
    {
      for (i = 0; i < (buf_size+1); i++)
      {
        temp_euc[i] = data_read[i].euc_data[j];
      }
    }
    else
    {
      for (i = 0; i < buf_size; i++)
      {
        temp_euc[i] = data_read[i].euc_data[j];
      }
    }

    /* Determine max loads Vmax and min loads Vmin */

    i = 0;
    index = 0;

    while ((i < buf_size) && (processed_done[j] == FALSE))
    {
      if (i == 0)
      {
        if (first_time == TRUE)
        {
          l = 0;

          if (temp_euc[i] < temp_euc[i+1])
          {
            prev_type[j] = MIN;
  prev_value[j] = temp_euc[i];
  min_max_value[j][l] = temp_euc[i];
  min_max_time[j][l] = data_read[i].irig_time;
  index = i+1;
}
else if (temp_euc[i] > temp_euc[i+1])
{
  prev_type[j] = MAX;
  prev_value[j] = temp_euc[i];
  min_max_value[j][l] = temp_euc[i];
  min_max_time[j][l] = data_read[i].irig_time;
  index = i+1;
}
else
{
  max_min_found = FALSE;

  for (k = (i+1); k < buf_size; k++)
  {
    if (temp_euc[k] > temp_euc[k+1])
    {
      max_min_found = TRUE;
      prev_type[j] = MAX;
      prev_value[j] = temp_euc[k];
      min_max_value[j][l] = temp_euc[k];
      min_max_time[j][l] = data_read[k].irig_time;
      index = k+1;
      break;
    }
    else if (temp_euc[k] < temp_euc[k+1])
    {
      max_min_found = TRUE;
      prev_type[j] = MIN;
      prev_value[j] = temp_euc[k];
      min_max_value[j][l] = temp_euc[k];
      min_max_time[j][l] = data_read[k].irig_time;
      index = k+1;
      break;
    }
  } /* for (k = (i+1); k < buf_size; k++) */

  if (max_min_found == FALSE)
  {
    printf("\nChan %s has the same data value  of %15.2f!!!\n",
            chan_name[j], temp_euc[1]);
    printf("crackGrowth program terminated!!!\n");
```

```
      exit (0);
    }
  } /* if (temp_euc[i] < temp_euc[i+1]) */
} /* if (first_time == TRUE) */
else
{
  l = 1;
  min_max_value[j][0] = last_data[j];
  min_max_time[j][0] = last_data_time[j];

  if ((prev_value[j] == end_value[j]) &&  (prev_type[j] == end_type[j]))
  {
    if (temp_euc[i] < temp_euc[i+1])
    {
      if (prev_type[j] == MAX)
      {
        prev_type[j] = MIN;
        prev_value[j] = temp_euc[i];
        min_max_value[j][l] = temp_euc[i];
        min_max_time[j][l] = data_read[i].irig_time;
        index = i+1;
      }
      else
      {
        for (k = (i+1); k < buf_size; k++)
        {
          if (temp_euc[k] > temp_euc[k+1])
          {
            prev_type[j] = MAX;
            prev_value[j] = temp_euc[k];
            min_max_value[j][l] = temp_euc[k];
            min_max_time[j][l] = data_read[k].irig_time;
            index = k+1;
            break;
          }
        }
      }
    }
    else
    {
      if (prev_type[j] == MIN)
      {
        prev_type[j] = MAX;
        prev_value[j] = temp_euc[i];
        min_max_value[j][l] = temp_euc[i];
        min_max_time[j][l] = data_read[i].irig_time;
      index = i+1;
    }
    else
    {
      for (k = (i+1); k < buf_size; k++)
      {
        if (temp_euc[k] < temp_euc[k+1])
        {
          prev_type[j] = MIN;
          prev_value[j] = temp_euc[k];
          min_max_value[j][l] = temp_euc[k];
          min_max_time[j][l] = data_read[k].irig_time;
          index = k+1;
          break;
        }
      } /* for (k = (i+1); k < buf_size; k++) */
    } /* else */
  }
}
else
{
  if (temp_euc[i] < temp_euc[i+1])
  {
    if (temp_euc[i] < end_value[j])
    {
      prev_type[j] = MIN;
      prev_value[j] = temp_euc[i];
      min_max_value[j][l] = temp_euc[i];
      min_max_time[j][l] = data_read[i].irig_time;
      index = i+1;
    }
    else
    {
      for (k = (i+1); k < buf_size; k++)
      {
        if (temp_euc[k] > temp_euc[k+1])
        {
          prev_type[j] = MAX;
          prev_value[j] = temp_euc[k];
          min_max_value[j][l] = temp_euc[k];
          min_max_time[j][l] = data_read[k].irig_time;
          index = k+1;
          break;
        }
      } /* for (k = (i+1); k < buf_size; k++) */
    }
```

```
      }/* if (temp_euc[i] < temp_euc[i+1]) */
      else
      {
        if (temp_euc[i] > end_value[j])
        {
          prev_type[j] = MAX;
          prev_value[j] = temp_euc[i];
          min_max_value[j][l] = temp_euc[i];
          min_max_time[j][l] = data_read[i].irig_time;
          index = i+1;
        }
        else
        {
          for (k = (i+1); k < buf_size; k++)
          {
            if (temp_euc[k] < temp_euc[k+1])
            {
              prev_type[j] = MIN;
              prev_value[j] = temp_euc[k];
              min_max_value[j][l] = temp_euc[k];
              min_max_time[j][l] = data_read[k].irig_time;
              index = k+1;
              break;
            } /* if (temp_euc[k] < temp_euc[k+1]) */
          } /* for (k = (i+1); k < buf_size; k++) */
        }
      }
    }
  } /* first_time == FALSE */

  fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
         (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
  l++;
}
else if (i <= (buf_size-2))
{
  if (temp_euc[i] > temp_euc[i+1])
  {
    if (prev_type[j] == MIN)
    {
      prev_type[j] = MAX;
      prev_value[j] = temp_euc[i];
      min_max_value[j][l] = temp_euc[i];
      min_max_time[j][l] = data_read[i].irig_time;
      index = i+1;
  fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
          (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
  l++;
}
else
{
if ((i+1) >= (buf_size-1))
{
  processed_done[j] = TRUE;

  if (last_data_read == TRUE)
  {
    if (min_max_value[j][l-1] != temp_euc[buf_size-1])
    {
      min_max_value[j][l] = temp_euc[buf_size-1];
      min_max_time[j][l] = data_read[buf_size-1].irig_time;
      fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
              (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
      l++;
    }
  }
  else
  {
    if (temp_euc[buf_size-1] < temp_euc[buf_size])
    {
      prev_type[j] = MIN;
      end_type[j] = MIN;
      prev_value[j] = temp_euc[buf_size-1];
      end_value[j] = temp_euc[buf_size-1];
      min_max_value[j][l] = temp_euc[buf_size-1];
      min_max_time[j][l] = data_read[buf_size-1].irig_time;
      fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
              (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);

      l++;
    }
    else
    {
      end_type[j] = NONE;
      end_value[j] = temp_euc[buf_size-1];
    }
  }
  index = buf_size;
}
else
{
```

```
for (k = (i+1); k < buf_size; k++)
{
  if (k < (buf_size-1))
  {
    if (temp_euc[k] < temp_euc[k+1])
    {
      prev_type[j] = MIN;
      min_max_value[j][l] = temp_euc[k];
      min_max_time[j][l] = data_read[k].irig_time;
      fprintf(fp_sigma_txt[j],  " %9.3f    %8.2f    \n",
           (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
      l++;
      index = k+1;
      break;
    }
  }
  else
  {
    processed_done[j] = TRUE;

    if (last_data_read == TRUE)
    {
      if (min_max_value[j][l-1] != temp_euc[buf_size-1])
      {
        min_max_value[j][l] = temp_euc[buf_size-1];
        min_max_time[j][l] = data_read[buf_size-1].irig_time;
        fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
               (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
        l++;
      }
    } /* if (last_data_read == TRUE) */
    else
    {
      if (temp_euc[buf_size-1] < temp_euc[buf_size])
      {
        prev_type[j] = MIN;
        end_type[j] = MIN;
        prev_value[j] = temp_euc[buf_size-1];
        end_value[j] = temp_euc[buf_size-1];
        min_max_value[j][l] = temp_euc[buf_size-1];
        min_max_time[j][l] = data_read[buf_size-1].irig_time;
        fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
               (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
        l++;
      }
      else
```

```
            {
              end_type[j] = NONE;
              end_value[j] = temp_euc[buf_size-1];
            }
          }
          index = buf_size;
        }
      }
    }
  }
}
else if (temp_euc[i] < temp_euc[i+1])
{
  if (prev_type[j] == MAX)
  {
    prev_type[j] = MIN;
    prev_value[j] = temp_euc[i];
    min_max_value[j][l] = temp_euc[i];
    min_max_time[j][l] = data_read[i].irig_time;
    fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
           (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
    index = i+1;
    l++;
    }
    else
    {
      if ((i+1) >= (buf_size-1))
      {
        processed_done[j] = TRUE;

        if (last_data_read == TRUE)
        {
          if (min_max_value[j][l-1] != temp_euc[buf_size-1])
          {
            min_max_value[j][l] = temp_euc[buf_size-1];
            min_max_time[j][l] = data_read[buf_size-1].irig_time;
            fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
                   (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
            l++;
          }
        } /* if (last_data_read == TRUE) */
        else
        {
          if (temp_euc[buf_size-1] > temp_euc[buf_size])
          {
            prev_type[j] = MAX;
      end_type[j] = MAX;
      prev_value[j] = temp_euc[buf_size-1];
      end_value[j] = temp_euc[buf_size-1];
      min_max_value[j][l] = temp_euc[buf_size-1];
      min_max_time[j][l] = data_read[buf_size-1].irig_time;
      fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
             (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
      l++;
    }
    else
    {
      end_type[j] = NONE;
      end_value[j] = temp_euc[buf_size-1];
    }
  }
  index = buf_size;
}
else
{
  for (k = (i+1); k < buf_size; k++)
  {
    if (k < (buf_size-1))
    {
      if (temp_euc[k] > temp_euc[k+1])
      {
        prev_type[j] = MAX;
        prev_value[j] = temp_euc[k];
        min_max_value[j][l] = temp_euc[k];
        min_max_time[j][l] = data_read[k].irig_time;
        fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
               (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
        l++;
        index = k+1;
        break;
      }
    }
    else
    {
      processed_done[j] = TRUE;

      if (last_data_read == TRUE)
      {
        if (min_max_value[j][l-1] != temp_euc[buf_size-1])
        {
          min_max_value[j][l] = temp_euc[buf_size-1];
          min_max_time[j][l] = data_read[buf_size-1].irig_time;
```

```
                fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
                       (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
                l++;
              }
            } /* if (last_data_read == TRUE) */
            else
            {
              if (temp_euc[buf_size-1] > temp_euc[buf_size])
              {
                prev_type[j] = MAX;
                end_type[j] = MAX;
                prev_value[j] = temp_euc[buf_size-1];
                end_value[j] = temp_euc[buf_size-1];
                min_max_value[j][l] = temp_euc[buf_size-1];
                min_max_time[j][l] = data_read[buf_size-1].irig_time;
                fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
                       (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
                l++;
              } /* if (temp_euc[buf_size-1] > temp_euc[buf_size]) */
              else
              {
                end_type[j] = NONE;
                end_value[j] = temp_euc[buf_size-1];
              }
            } /* if (last_data_read != TRUE) */
            index = buf_size;
          }
        }
      }
    }
  }
  else /* temp_euc[i] = temp_euc[i+1] */
  {
    index = i + 1;
  }
}
else if (i == (buf_size-1))
{
  processed_done[j] = TRUE;

  if (last_data_read == TRUE)
  {
    if (min_max_value[j][l-1] != temp_euc[buf_size-1])
    {
      min_max_value[j][l] = temp_euc[buf_size-1];
      min_max_time[j][l] = data_read[buf_size-1].irig_time;
    fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
           (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
    l++;
  }
} /* if (last_data_read == TRUE) */
else
{
  if (temp_euc[i] < temp_euc[i-1])
  {
    if (temp_euc[i] < temp_euc[i+1])
    {
      prev_type[j] = MIN;
      prev_value[j] = temp_euc[i];
      end_type[j] = MIN;
      end_value[j] = temp_euc[i];
      min_max_value[j][l] = temp_euc[i];
      min_max_time[j][l] = data_read[i].irig_time;
      fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
             (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
      l++;
    }
    else
    {
      end_type[j] = NONE;
      end_value[j] = temp_euc[i];
    }
  }
  else
  {
    if (temp_euc[i] > temp_euc[i+1])
    {
      prev_type[j] = MAX;
      prev_value[j] = temp_euc[i];
      end_type[j] = MAX;
      end_value[j] = temp_euc[i];
      min_max_value[j][l] = temp_euc[i];
      min_max_time[j][l] = data_read[i].irig_time;
      fprintf(fp_sigma_txt[j], " %9.3f    %8.2f    \n",
             (float) min_max_time[j][l]/10000.0, min_max_value[j][l]);
      l++;
    }
    else
    {
      end_type[j] = NONE;
      end_value[j] = temp_euc[i];
    }
```

```
            }
          } /* if (last_data_read != TRUE) */
          index = buf_size;
        }

        if (min_max_value[j][l-1] <= 0.0)
        {
          strncpy(ch_name, chan_name[j], 10);
          printf("\nChan %10s has BAD data %9.2f!! ", ch_name, min_max_value[j][l-1]);
          printf("at Irigb time = %9.3f\n\n", (float) min_max_time[j][l-1]/10000.0);
          printf("crackGrowth program terminated!!!\n");
          exit (0);
        }
        i = index;
      } /* while ((i < buf_size) && (processed_done[j] == FALSE)) */

      write_index[j] = l;
      last_data[j] = min_max_value[j][l-1];
      last_data_time[j] = min_max_time[j][l-1];
    } /* if (do_calc[j] == TRUE) */
  } /* for ( j = 0; j < num_chans; j++) */
}


/******************************************************************************
 * Subroutine calculate_deltaa()                                              *
 *                                                                            *
 * Description:                                                               *
 *   This subroutine uses the Vmax and Vmin loads determined in subroutine    *
 *   determine_max_min() to calculate half cycle crack growth delta a.  It sums up all *
 *   calculated delta a to get the total crack growth size. It also determines the worst half *
 *   cycle ratio Ri = load_min / load_max during the period between takeoff run and cruise *
 *   power. The worst half cycle Vmax load will be used in calculating the number of *
 *   operational flights.                                                     *
 *                                                                            *
 ******************************************************************************/
void calculate_deltaa()
{
  int       i, j, k, l;
  double    Ai_1, pow_Kmax, pow_Ri, sqrt_value, double_const;
  float     Ri, Kmax, float_sum_deltaa;
  float     load_max, load_min, stress;
  short     cal_deltaa;

  bzero((char *) (&deltaa), sizeof(deltaa));
  double_const = 10000.0;
```

```
/* For each channel */

for ( j = 0; j < num_chans; j++)
{
  if (do_calc[j] == TRUE)
  {
    if (first_time == TRUE)
    {
      Ai_1 = apc[j];
      sum_a[j] = apc[j];
    }
    else
    {
      Ai_1 = sum_a[j];
    }

    for (i = 1; i < write_index[j]; i++)
    {
      cal_deltaa = FALSE;

      /* Determine the Vmax and Vmin in this half cycle */

      if (min_max_value[j][i-1] < min_max_value[j][i])
      {
        load_min = min_max_value[j][i-1];
        load_max = min_max_value[j][i];
        cal_deltaa = TRUE;
      }
      else if (min_max_value[j][i-1] > min_max_value[j][i])
      {
        load_min = min_max_value[j][i];
        load_max = min_max_value[j][i-1];
        cal_deltaa = TRUE;
      }

      if (cal_deltaa == TRUE)
      {
        Ri = load_min / load_max;

        /* Add the previous crack growth delta a to Ai_1 */

        Ai_1 += deltaa[j][i-1];

        /* Convert load into stress */
```

```
          stress = stress_coef[j]*load_max;

          /* Calculate crack growth delta a */

          sqrt_value = sqrt((double) PI*Ai_1/Q);
          Kmax = A*Mk*stress*(float) sqrt_value;
          pow_Kmax = pow((double) Kmax, (double) m[j]);
          pow_Ri   = pow((double) (1.0-Ri), (double) n[j]);
          deltaa[j][i] = C_OVER_2[j] * pow_Kmax * pow_Ri;

          /* Add the current crack growth to the sum of delta a */
          sum_deltaa[j] += deltaa[j][i];
          sum_a[j] += deltaa[j][i];

          /* Determine the worst half cycle */
          if ((min_max_time[j][i-1] >= takeoff_time) &&
              (min_max_time[j][i-1] <= cruise_time))
          {
            if (Ri < Ri_min[j])
            {
              Ri_min[j] = Ri;
              Vmax[j] = load_max;
              Vmin[j] = load_min;
              Tmax[j] = min_max_time[j][i-1];
            }
          }
        } /* if (cal_deltaa == TRUE) */
      } /* for (i = 1; i < write_index[j]; i++) */

      float_sum_deltaa = (float) double_const * sum_deltaa[j];
      data_write1.euc_data[j] = double_const * sum_deltaa[j];
    } /* if (do_calc[j] == TRUE) */
  } /* for ( j = 0; j < num_chans; j++) */

  /* Write data into output file */

  fwrite(&data_write1, data_buff_size, 1, fpout1);

  if (first_time == TRUE)
  {
    first_time = FALSE;
  }
}


/******************************************************************************
```

```
 * Subroutine calculate_flights()                                                    *
 *                                                                                   *
 * Description:                                                                      *
 *    This subroutine uses the worst half cycle Vmax for calculating the number of operational  *
 *    flights based on the first flight load data.                                   *
 *                                                                                   *
 *************************************************************************/
void calculate_flights()
{
  int         i, j;
  float       real_flight, remain;
  double      pow_f1, pow_a;

  for (j = 0; j < num_chans; j++)
  {
    if (do_calc[j] == TRUE)
    {
      f[j] = Vmax[j] / Vstar[j];
      pow_f1 = pow((double) f[j], (double) (m[j]-2.0));
      numerator[j] = 1.0 - pow_f1;
      pow_a  = pow((double) (1.0 + (sum_deltaa[j]/apc[j])), (double) (1.0-(m[j]/2.0)));
      denominator[j] = 1.0 - pow_a;
      real_flight = numerator[j]/denominator[j];
      remain = real_flight - (int) real_flight;

      if (remain >= 0.5)
      {
        int_flight[j] = (int) real_flight + 1;
      }
      else
      {
        int_flight[j] = (int) real_flight;
      }
    }
  }
}


/**************************************************************************
 * Subroutine generate_summary()                                                     *
 *                                                                                   *
 * Description:                                                                      *
 *    This subroutine generates a summary for all channels in the input file. The summary      *
 *    contains the size of total crack growth, the number of operational flights, the operational  *
 *    load factor, the worst half cycle Vmax, Vmin, and its IRIGB time. Additionally, the         *
 *    numerator and  the denominator that are used to calculate the number of operational          *
 *    flights are also included in the summary.                                                      *
 *                                                                                                    *
 *****************************************************************************/
void generate_summary()
{
  int         i, j;
  char        name[NAMES_SIZE];

  fwrite("B-52B hooks     Crack Growth  Flights    f        Vmax       Vmin       Irigb time\n\n",
   strlen("B-52B hooks     Crack Growth  Flights    f        Vmax       Vmin       Irigb time\n\n"),
          1, fpout2);

  for (j = 0; j < num_chans; j++)
  {
    if (do_calc[j] == TRUE)
    {
      strncpy(name, chan_name[j], NAMES_SIZE);
      fprintf(fpout2, "%s", name);
      fprintf(fpout2, "%10.4e  ", sum_deltaa[j]);
      fprintf(fpout2, "%6d  ", int_flight[j]);
      fprintf(fpout2, "  %7.4f ", f[j]);
      fprintf(fpout2, "  %9.2f   ", Vmax[j]);
      fprintf(fpout2, "%9.2f    ", Vmin[j]);
      fprintf(fpout2, "%9.3f\n", (float) Tmax[j]/10000.0);
    }
  }

  fwrite("\n\n\nB-52B hooks     Numerator     Denominator\n\n",
   strlen("\n\n\nB-52B hooks     Numerator     Denominator\n\n"), 1, fpout2);

  for (j = 0; j < num_chans; j++)
  {
    if (do_calc[j] == TRUE)
    {
      strncpy(name, chan_name[j], NAMES_SIZE);
      fprintf(fpout2, "%s", name);
      fprintf(fpout2, "%10.4e    ", numerator[j]);
      fprintf(fpout2, "%10.4e\n", denominator[j]);
    }
  }
}
```

APPENDIX C
MATERIAL PROPERTIES

Material properties of B-52B pylon hooks and Pegasus adapter pylon hooks are listed in Table C1 and Table C2.

Table C1. Material properties of B-52B pylon hooks and Pegasus adapter pylon hooks.

Component	Material	σ_U	σ_Y	τ_U	K_{IC}	C	m	n
		ksi	ksi	ksi	$\text{ksi}\sqrt{\text{in.}}$	$\frac{\text{in.}}{\text{cycle}}\left(\text{ksi}\sqrt{\text{in.}}\right)^{-\text{m}}$		
B-52B front hook	Inconel 718*	175	145	135	125	0.922×10^{-11}	3.60	2.16
B-52B rear hooks	AMAX MP35N^	250	235	141	124	2.944×10^{-11}	3.24	1.69
Pegasus hooks	AMAX MP35N^	250	235	141	124	2.944×10^{-11}	3.24	1.69

* Inconel 718 is a registered trademark of Huntington Alloy Products Division, International Nickel Company, West Virginia.

^ AMAX MP35N is a trademark of SPS Technologies, Inc., Jenkintown, Pennsylvania.

Table C2. Material properties of Inconel 718 and AMAX MP35Nalloys.

Material	E, lb/in^2	G, lb/in^2	ν	ρ, lb/in^3	α, in/in-°F
Inconel 718	29.60×10^6	-----	-----	0.297	6.40×10^6
AMAX MP35N	34.05×10^6	11.74×10^6	0.39	0.322	7.10×10^6

FIGURES

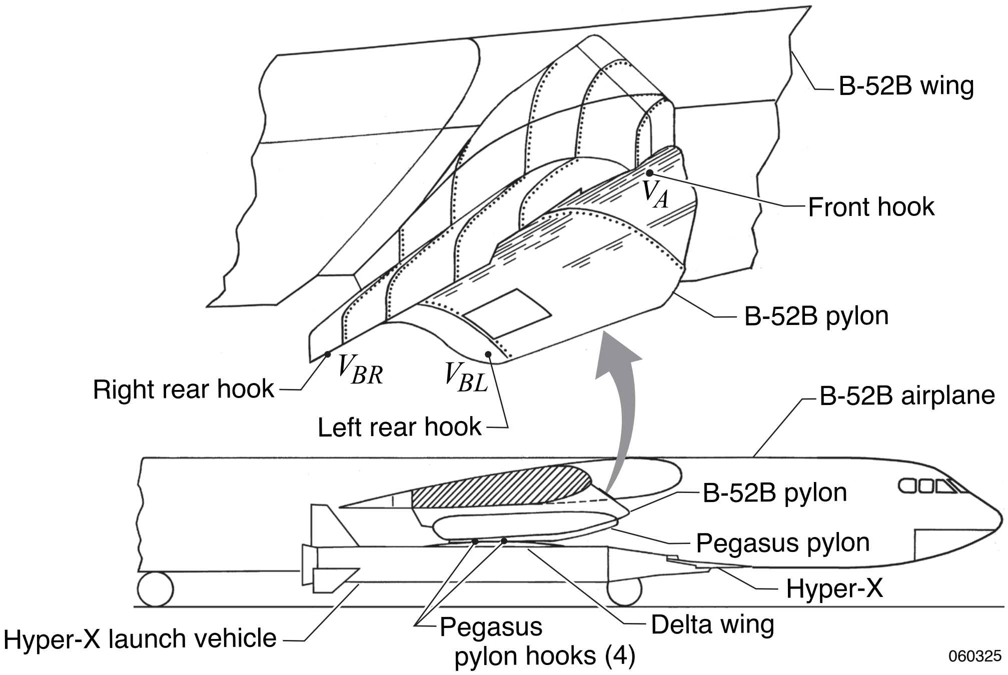
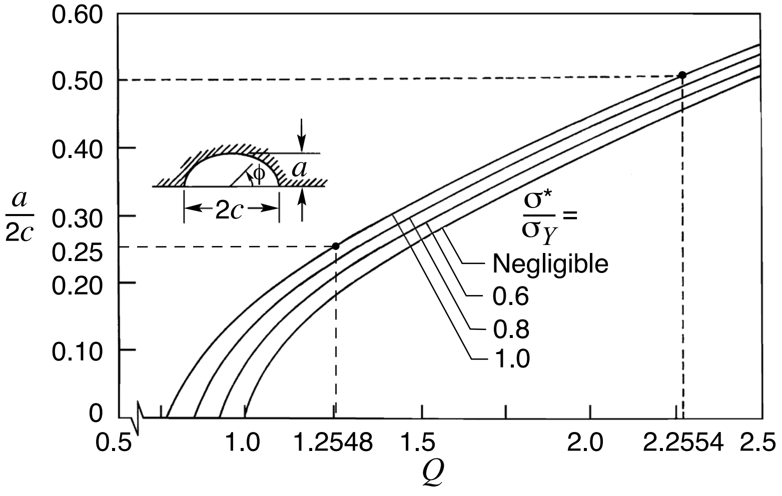


Figure 1. The B-52B airplane carrying the winged Pegasus rocket/X-43 systems (40,000 lb).



$$Q = [E(k)]^2 - 0.212 \left(\frac{\sigma^*}{\sigma_Y}\right)^2$$

$$E(k) = \int_0^{\frac{\pi}{2}} \sqrt{1 - k^2 \sin^2 \phi}\, d\phi \; ; \quad k = \sqrt{1 - \left(\frac{a}{c}\right)^2}$$

060326

Figure 2. Surface flaw shape and plasticity factor for semi-elliptic surface cracks.

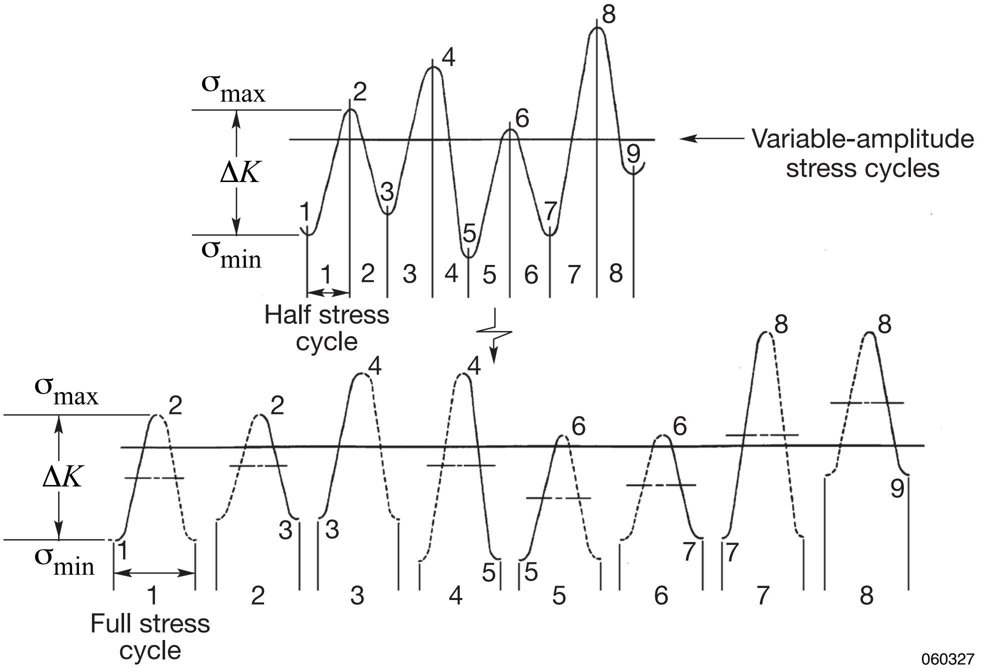


Figure 3. Resolution of random stress cycles into half stress cycles of different stress ranges.

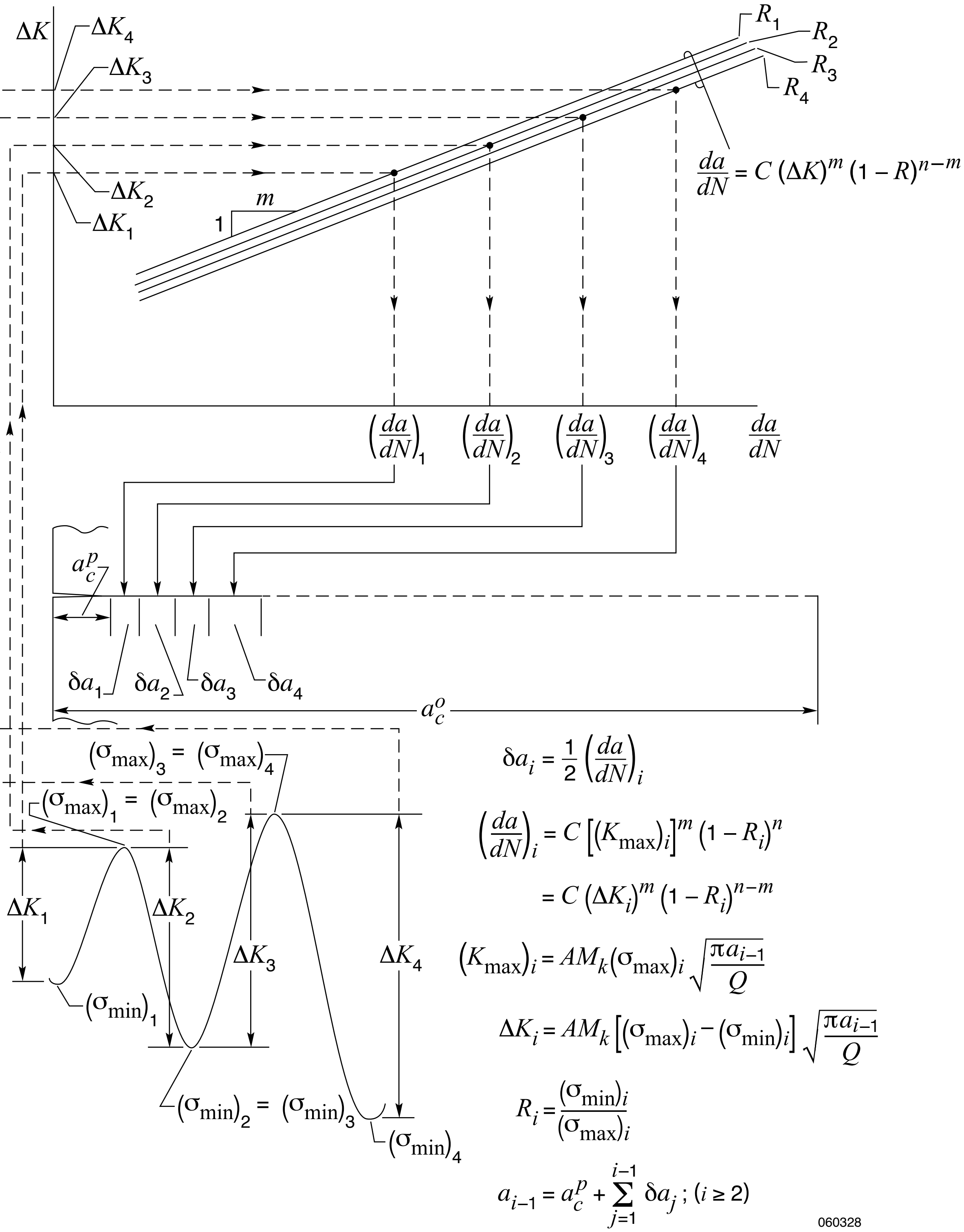

ΔK
ΔK_4
ΔK_3
ΔK_2
ΔK_1
R_1
R_2
R_3
R_4
m
1
$\frac{da}{dN} = C\,(\Delta K)^m\,(1 - R)^{n-m}$
$\left(\frac{da}{dN}\right)_1$
$\left(\frac{da}{dN}\right)_2$
$\left(\frac{da}{dN}\right)_3$
$\left(\frac{da}{dN}\right)_4$
$\frac{da}{dN}$
a_c^p
δa_1
δa_2
δa_3
δa_4
a_c^o
$(\sigma_{\max})_3 = (\sigma_{\max})_4$
$(\sigma_{\max})_1 = (\sigma_{\max})_2$
ΔK_1
ΔK_2
ΔK_3
ΔK_4
$(\sigma_{\min})_1$
$(\sigma_{\min})_2 = (\sigma_{\min})_3$
$(\sigma_{\min})_4$
$\delta a_i = \frac{1}{2}\left(\frac{da}{dN}\right)_i$
$\left(\frac{da}{dN}\right)_i = C\left[(K_{\max})_i\right]^m (1 - R_i)^n$
$= C\,(\Delta K_i)^m\,(1 - R_i)^{n-m}$
$(K_{\max})_i = AM_k(\sigma_{\max})_i\sqrt{\frac{\pi a_{i-1}}{Q}}$
$\Delta K_i = AM_k\left[(\sigma_{\max})_i - (\sigma_{\min})_i\right]\sqrt{\frac{\pi a_{i-1}}{Q}}$
$R_i = \frac{(\sigma_{\min})_i}{(\sigma_{\max})_i}$
$a_{i-1} = a_c^p + \sum_{j=1}^{i-1} \delta a_j\,;\ (i \geq 2)$
060328

Figure 4. Graphic evaluation of crack growths caused by random loading spectrum using the half-cycle theory.

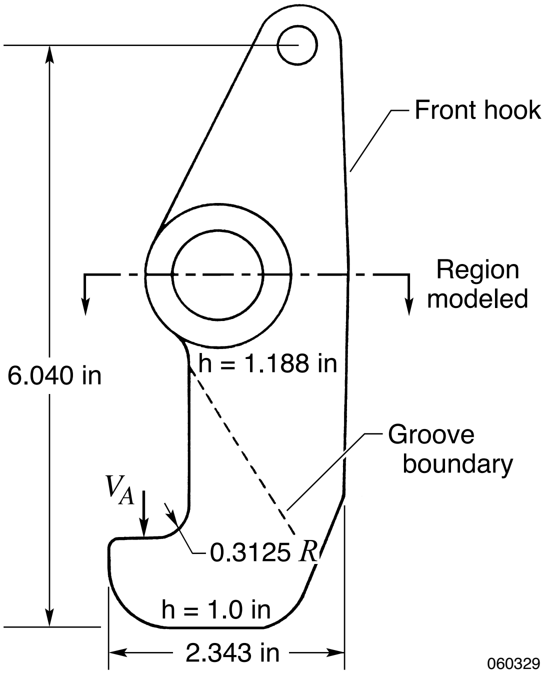


Figure 5. Geometry of B-52B pylon front hook.

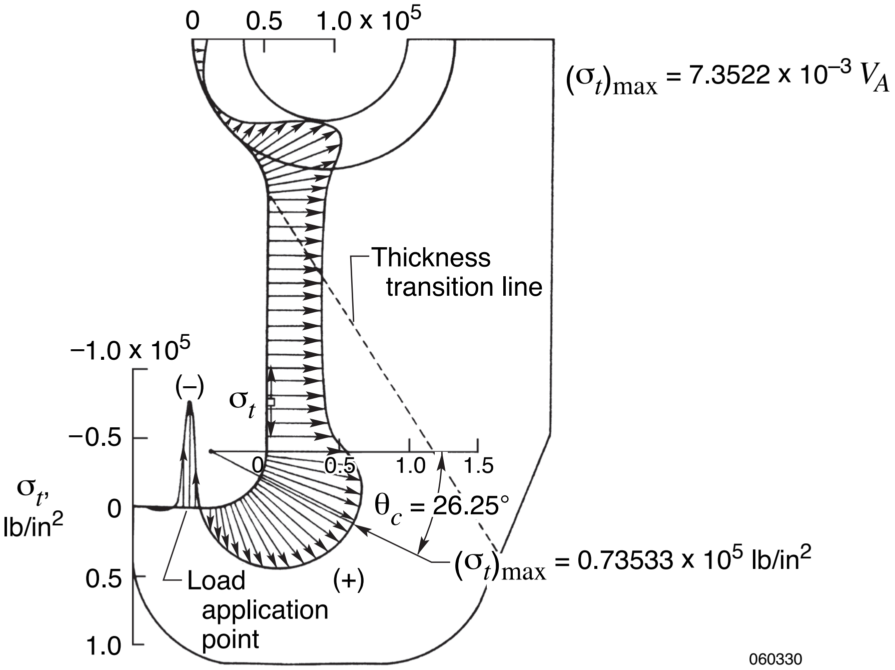


Figure 6. Distribution of tangential stress, σ_t , along the inner boundary of the B-52B pylon front hook; V_A =10,000 lb.

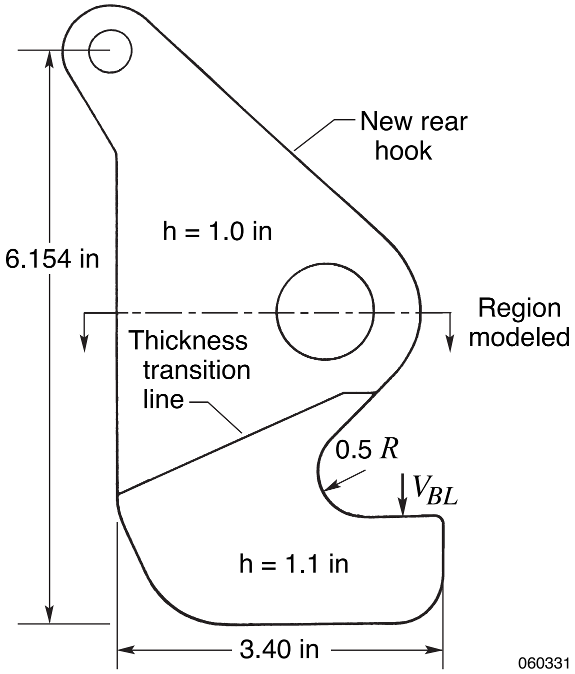


Figure 7. Geometry of the B-52B pylon rear hook.

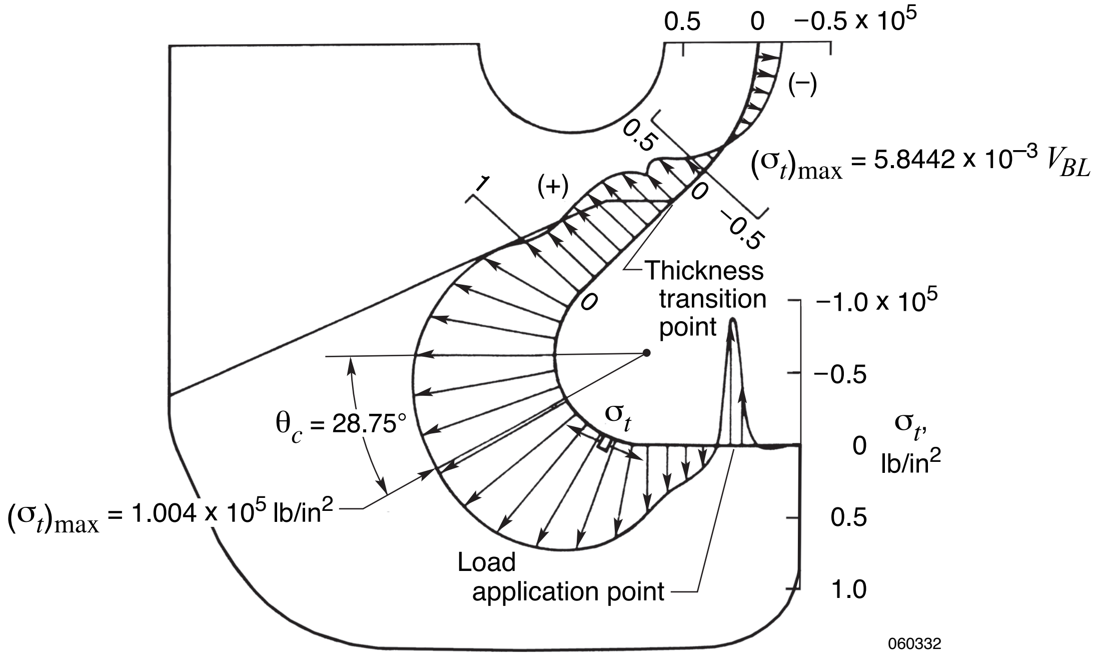


Figure 8. Distribtuion of tangential stress, σ_t , along the inner boundary of a typical B-52B pylon rear hook; V_{BL} =17,179.53 lb.

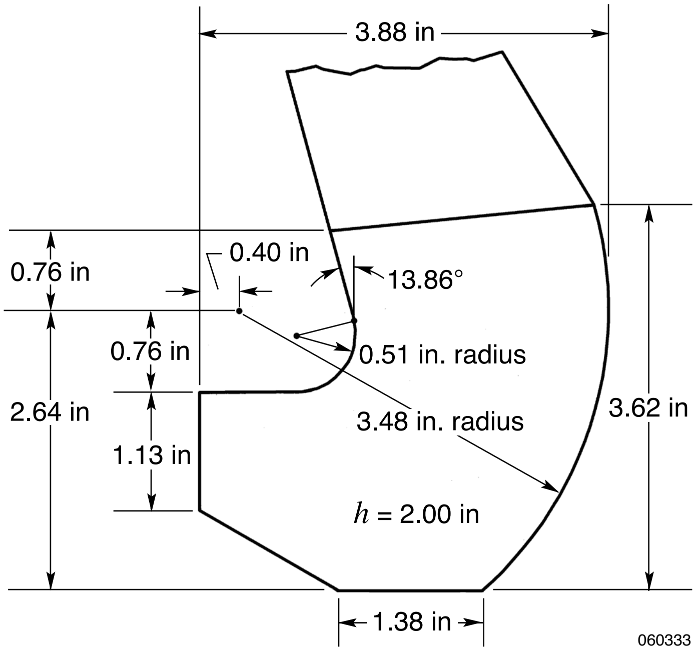


Figure 9. Geometry of the Pegasus pylon hook.

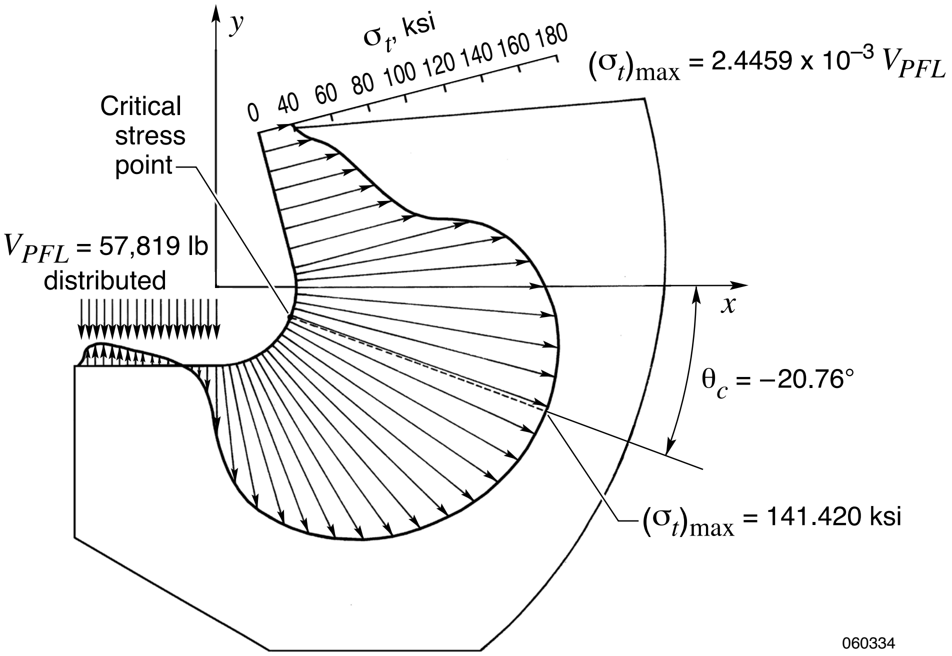

y
σ_t, ksi
0 40 60 80 100 120 140 160 180
$(\sigma_t)_{\max}$ = 2.4459 x 10^{-3} V_{PFL}
Critical stress point
V_{PFL} = 57,819 lb distributed
x
θ_c = −20.76°
$(\sigma_t)_{\max}$ = 141.420 ksi
060334

Figure 10. Distribution of tangential stress, σ_t , along the inner boundary of a typical Pegasus pylon hook; V_{PFL} =57,819 lb.

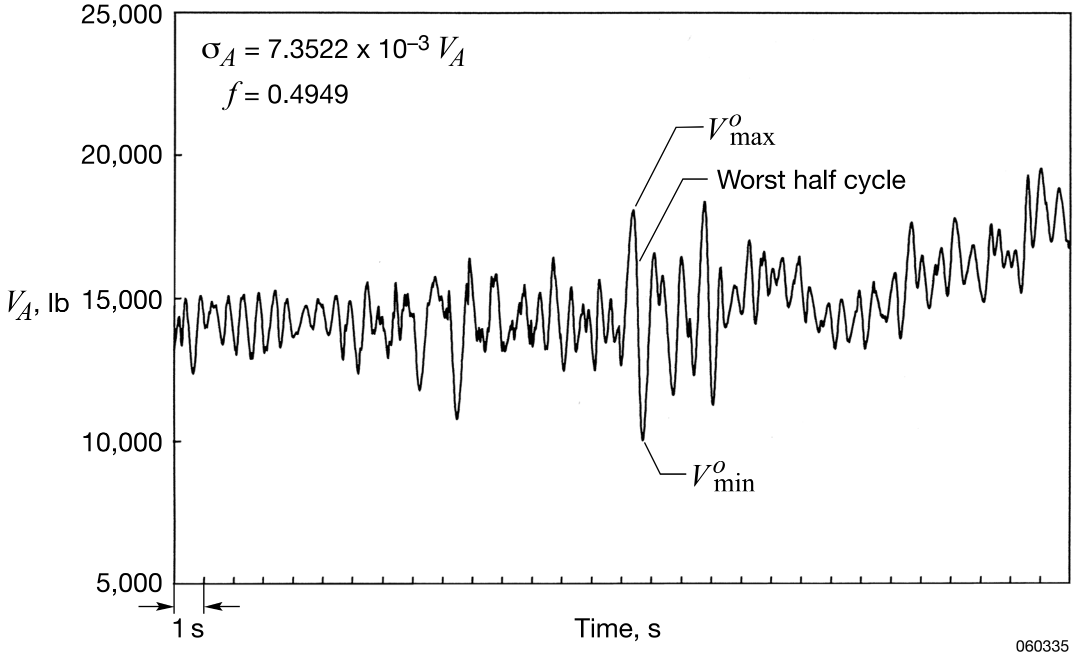


Figure 11. Loading spectrum of the B-52B front hook (VA) carrying the Hyper-X launching vehicle during takeoff.

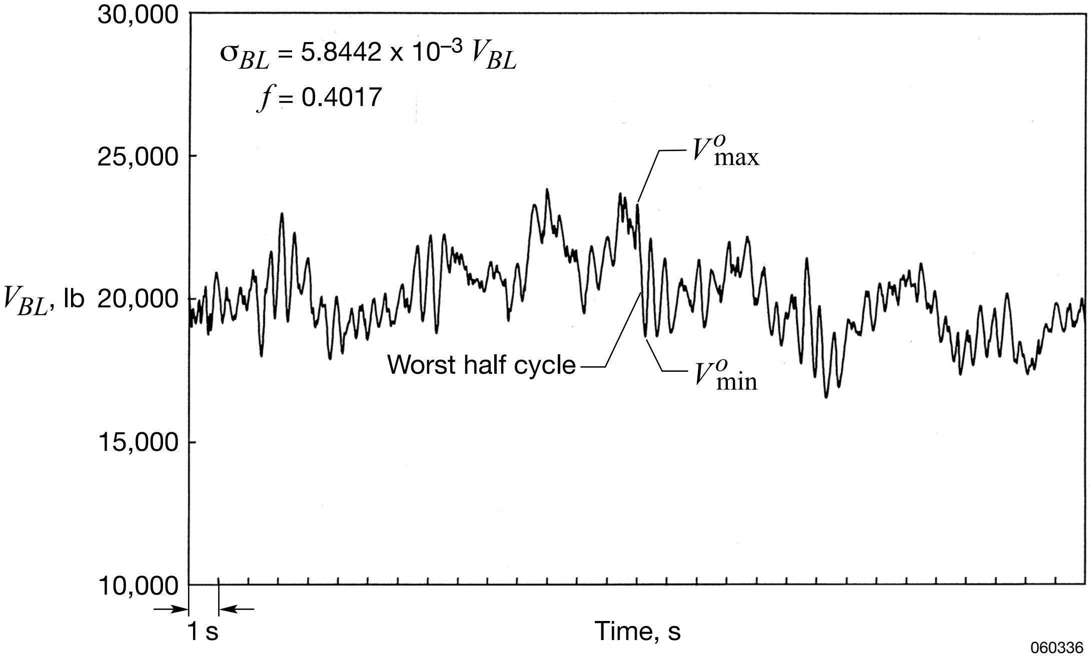


Figure 12. Loading spectrum of the B-52B rear left hook (VBL) carrying the Hyper-X launch vehicle during takeoff.

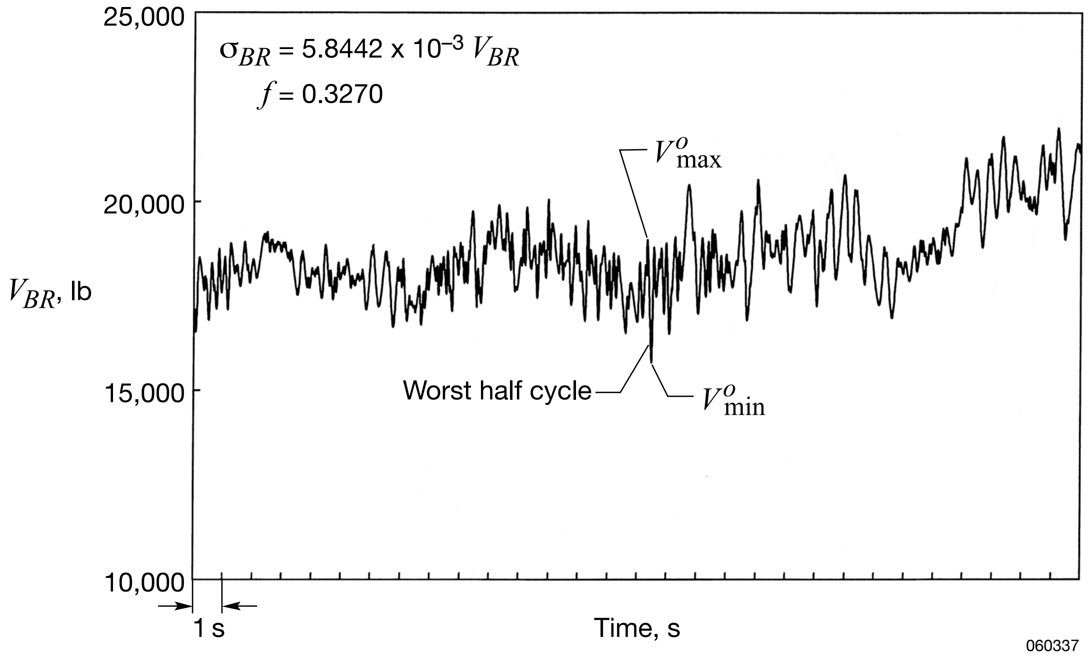


Figure 13. Loading spectrum of the B-52B rear right hook (VBR) carrying the Hyper-X launching vehicle during takeoff.

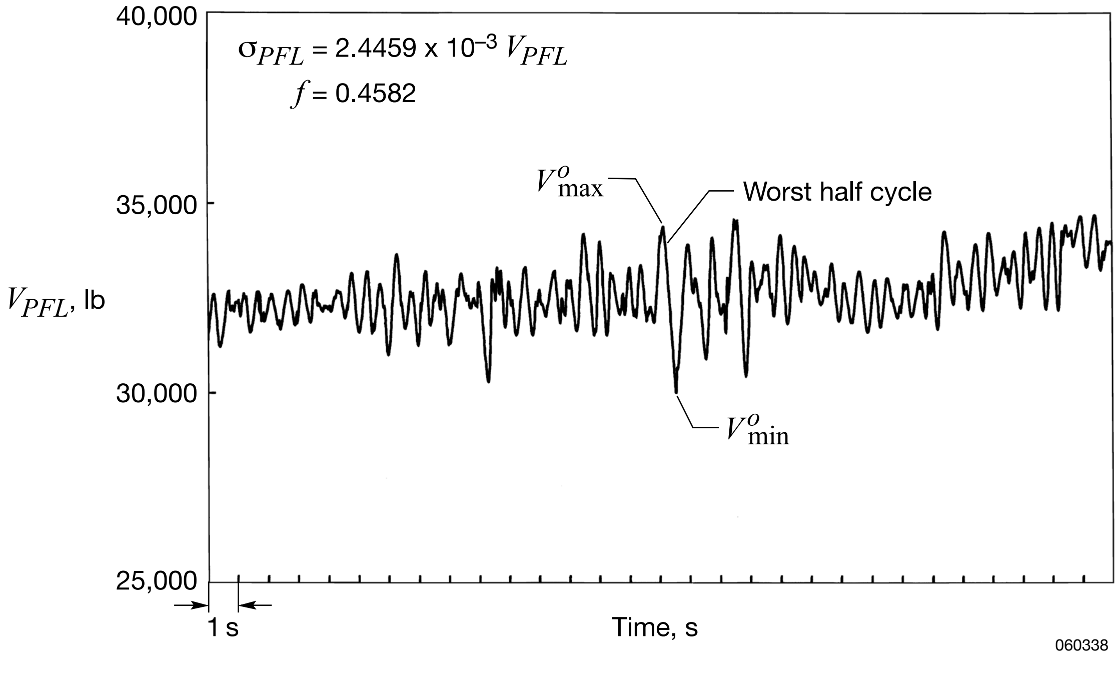


Figure 14. Loading spectrum of the Pegasus pylon front left hook (VPFL) carrying the Hyper-X launching vehicle during takeoff.

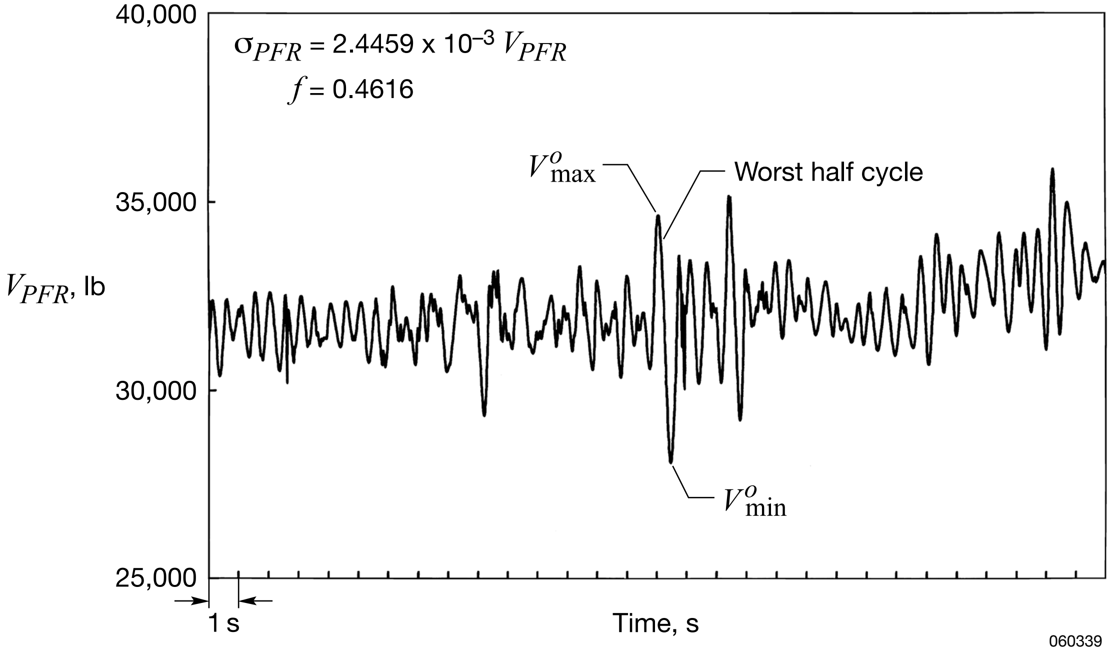


Figure 15. Loading spectrum of the Pegasus pylon front right hook (VPFR) carrying the Hyper-X launching vehicle during takeoff.

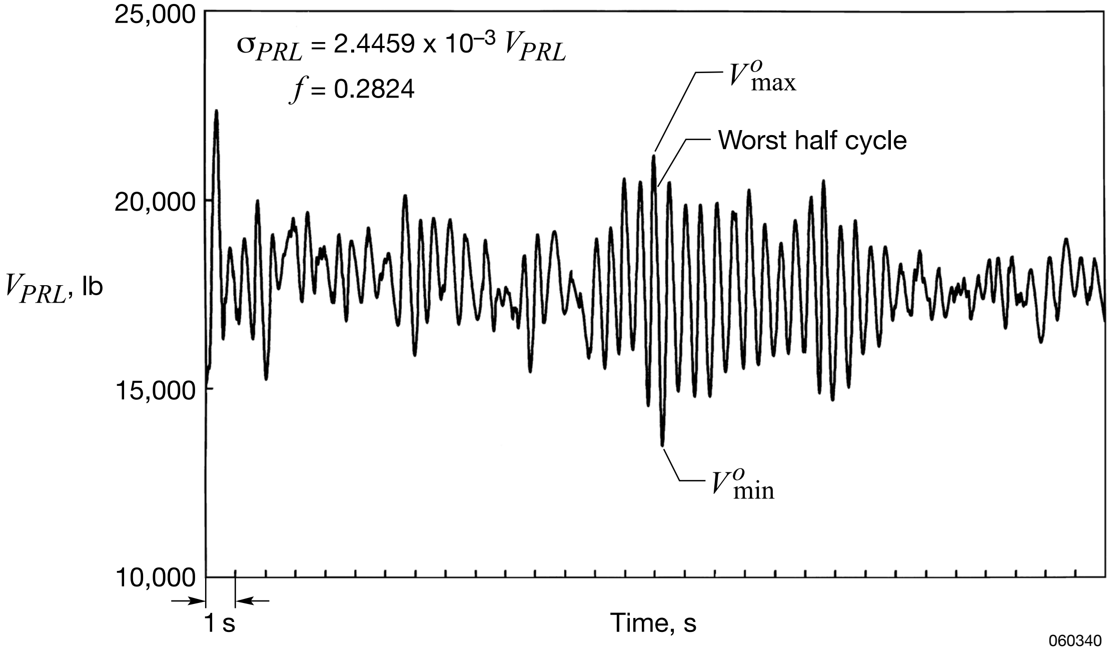


Figure 16. Loading spectrum of the Pegasus pylon rear left hook (VPRL) carrying Hyper-X launching vehicle during takeoff.

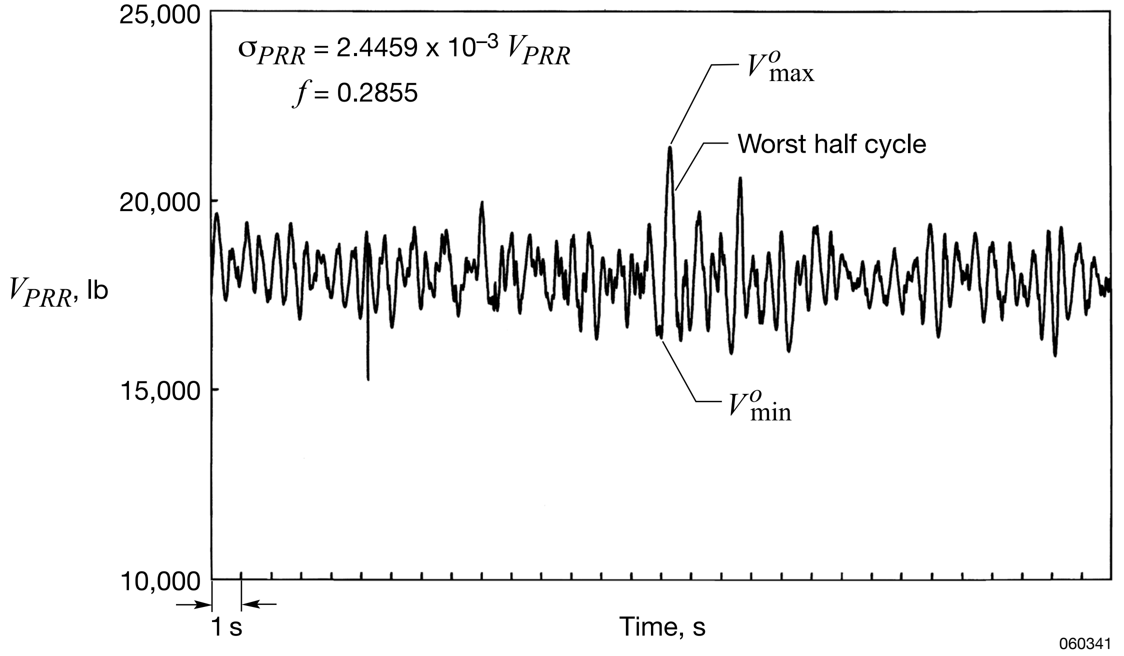


Figure 17. Loading spectrum of the Pegasus pylon rear right hook (VPRR) carring the Hyper-X launching vehicle during takeoff.

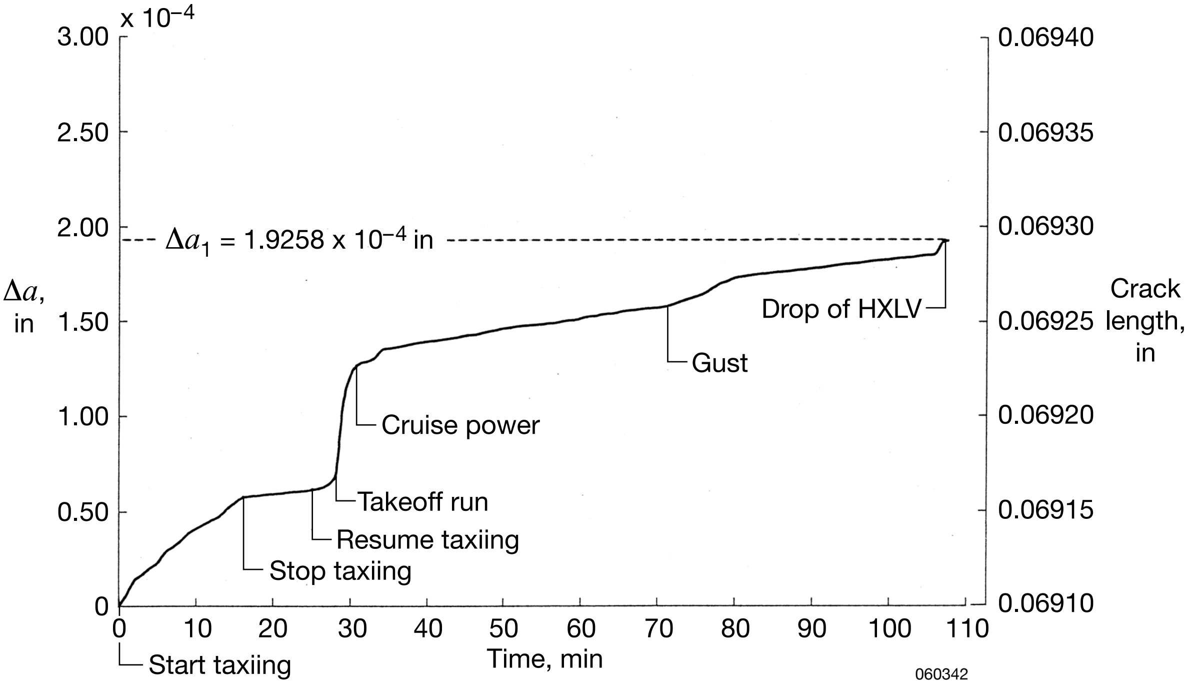


Figure 18. Crack growth curve for the B-52B front hook (VA); B-52B carrying the Hyper-X launching vehicle; air-launching flight.

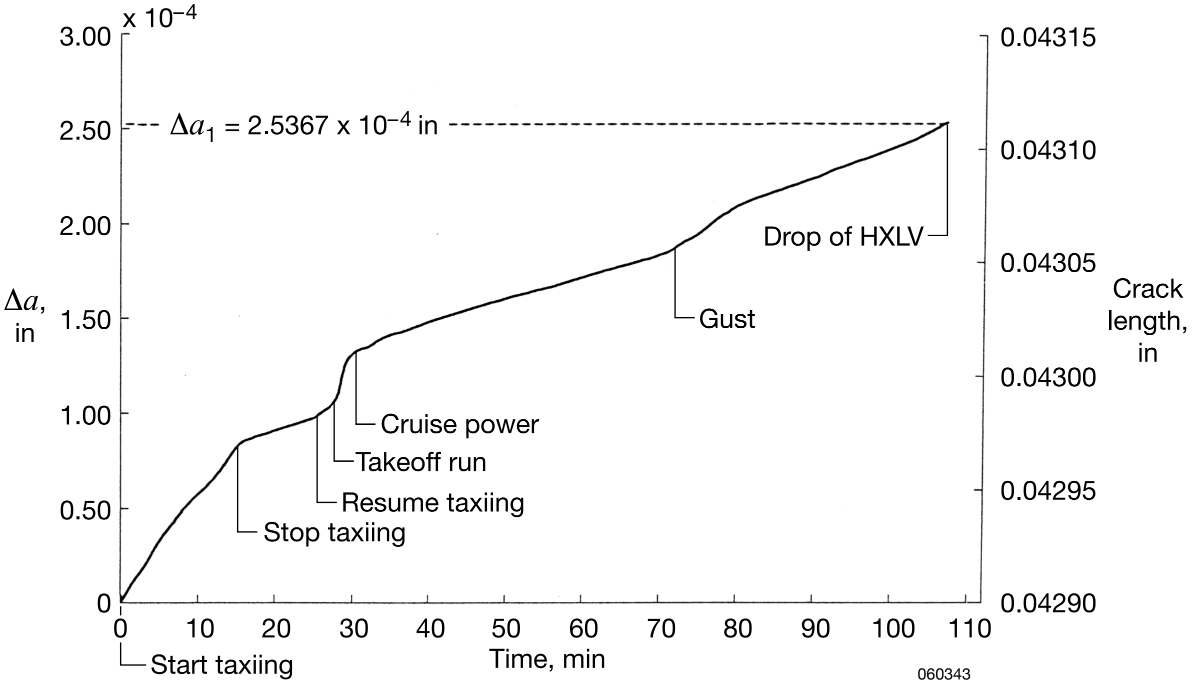


Figure 19. Crack growth curve for the B-52B rear left hook (VBL); B-52B carrying the Hyper-X launching vehicle; air-launching flight.

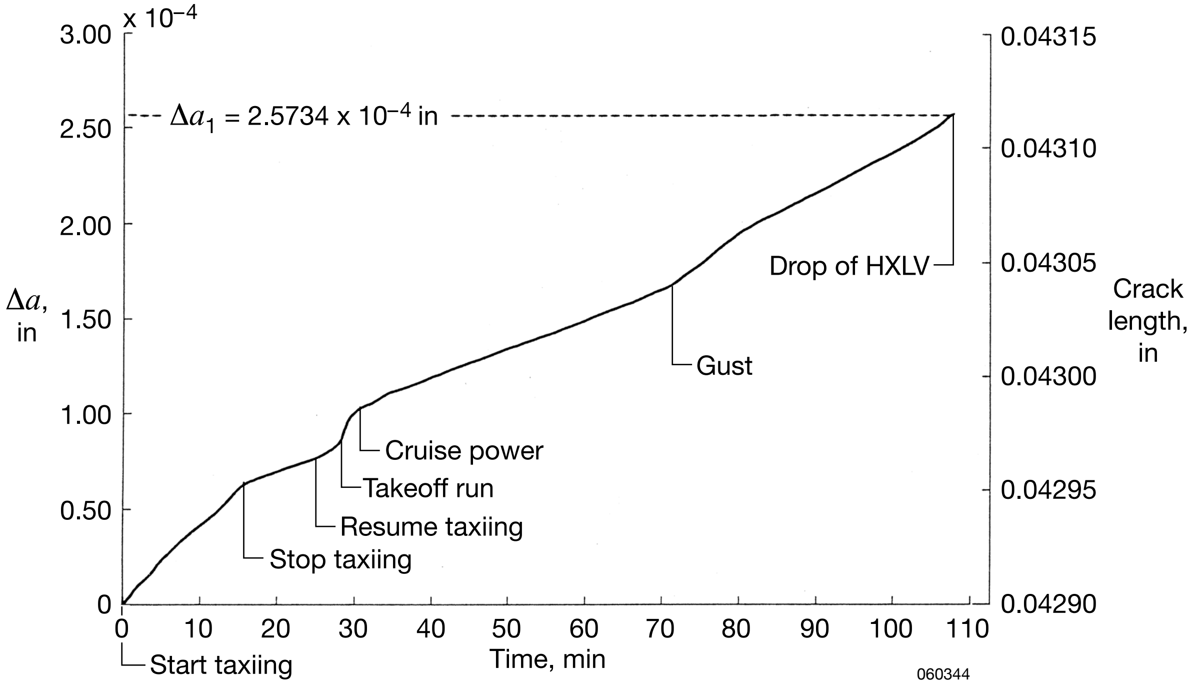


Figure 20. Crack growth curve for the B-52B rear right hook (VBR); B-52B carrying the Hyper-X launching vehicle; air-launching flight.

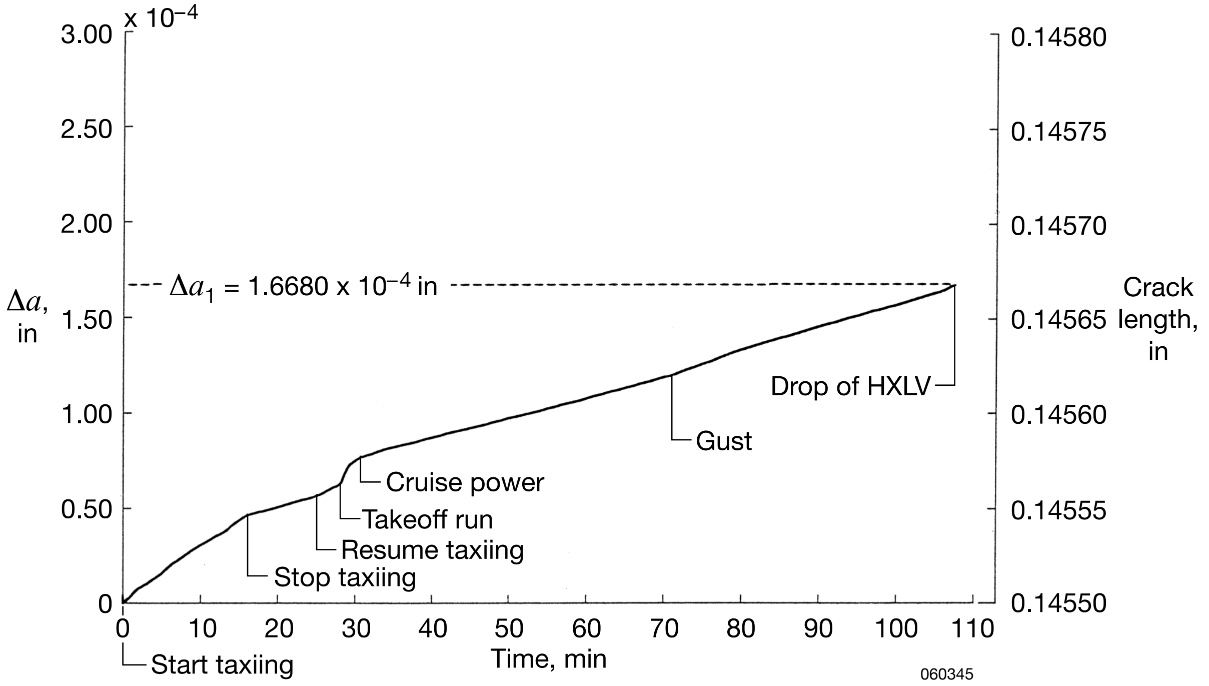


Figure 21. Crack growth curve for the Pegasus pylon front left hook (VPFL); B-52B carrying the Hyper-X launching vehicle; air-launching flight.

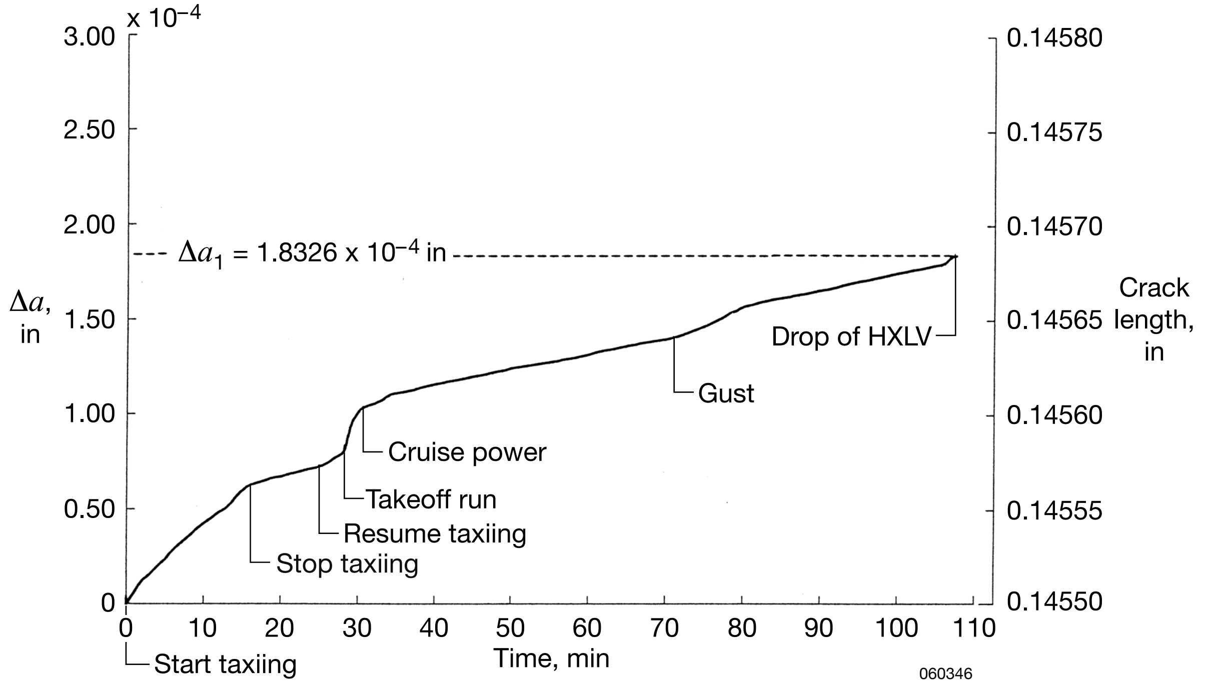


Figure 22. Crack growth curve for the Pegasus pylon front right hook (VPFR); B-52B carrying the Hyper-X launching vehicle; air-launching flight.

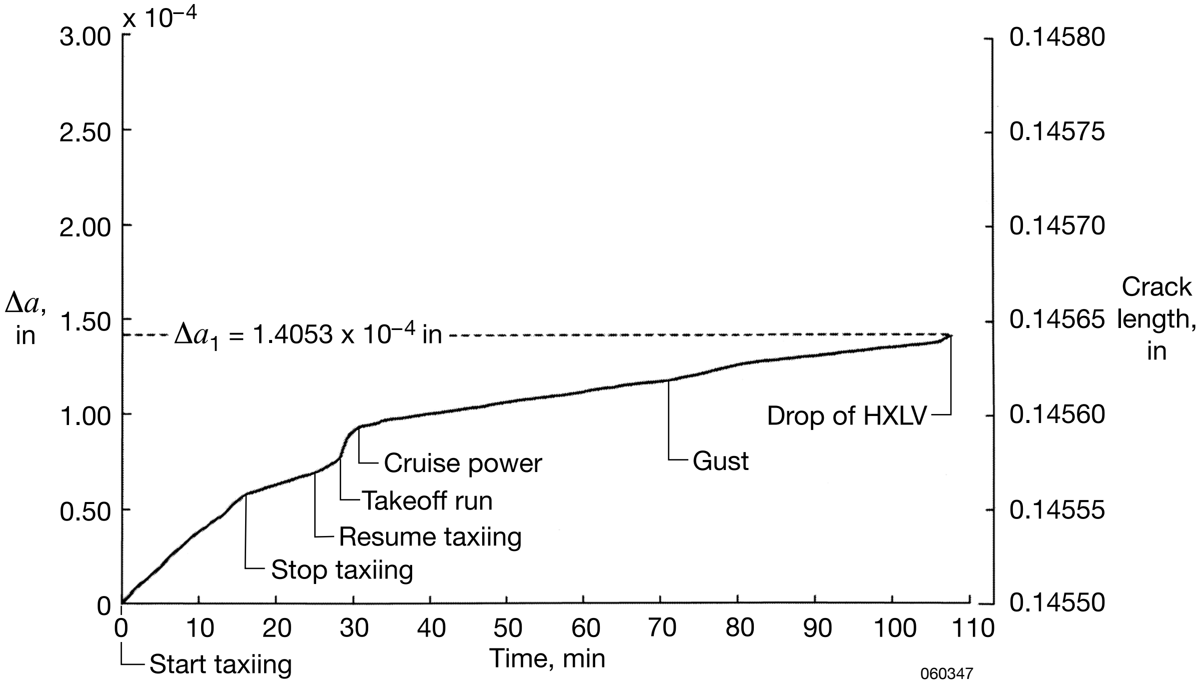


Figure 23. Crack growth curve for the Pegasus pylon rear left hook (VPRL); B-52B carrying the Hyper-X launching vehicle; air-launching flight.

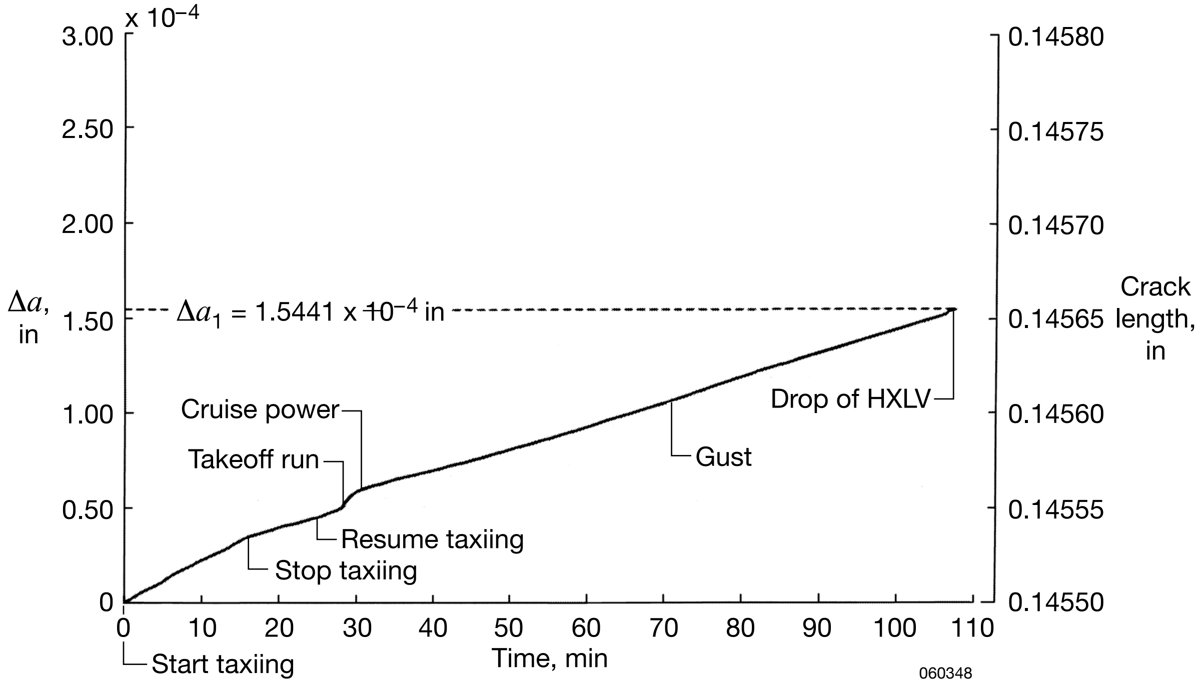


Figure 24. Crack growth curve for the Pegasus pylon rear right hook (VPRR); B-52B carrying the the Hyper-X launching vehicle; air-launching flight.

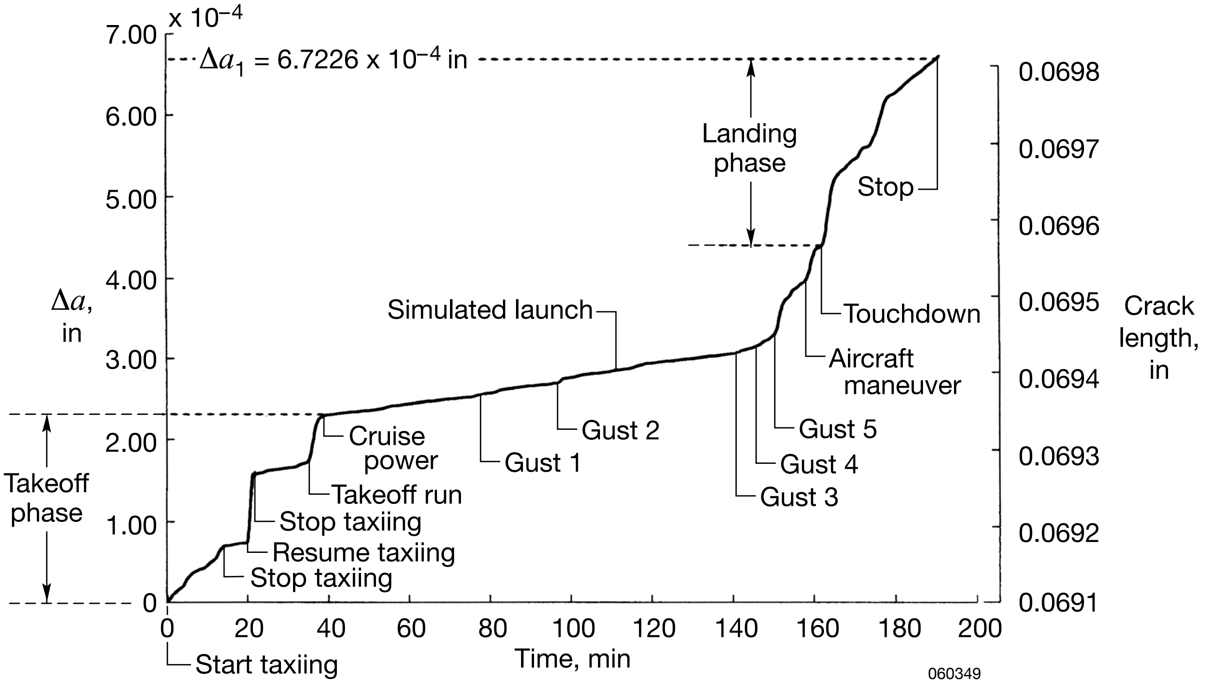


Figure 25. Crack growth curve for the Pegasus pylon front hook (VA); B-52B carrying the Hyper-X launching vehicle; captive flight.

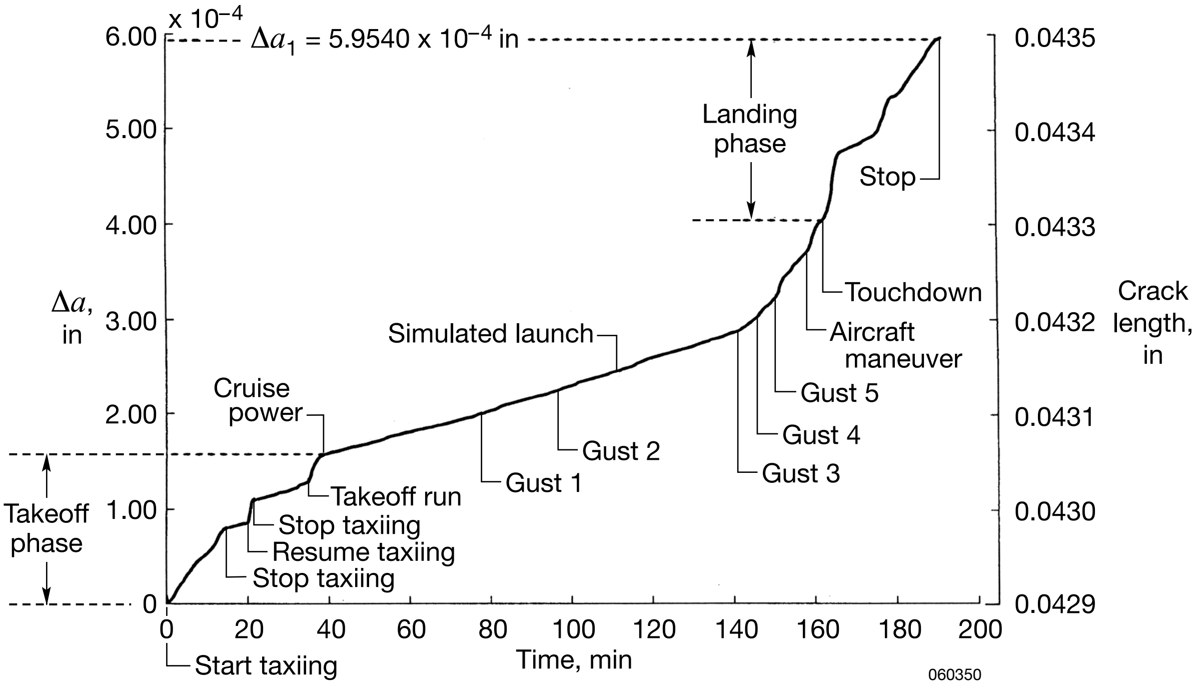


Figure 26. Crack growth curve for the B-52B rear left hook (VBL); B-52B carrying the Hyper-X launching vehicle; captive flight.

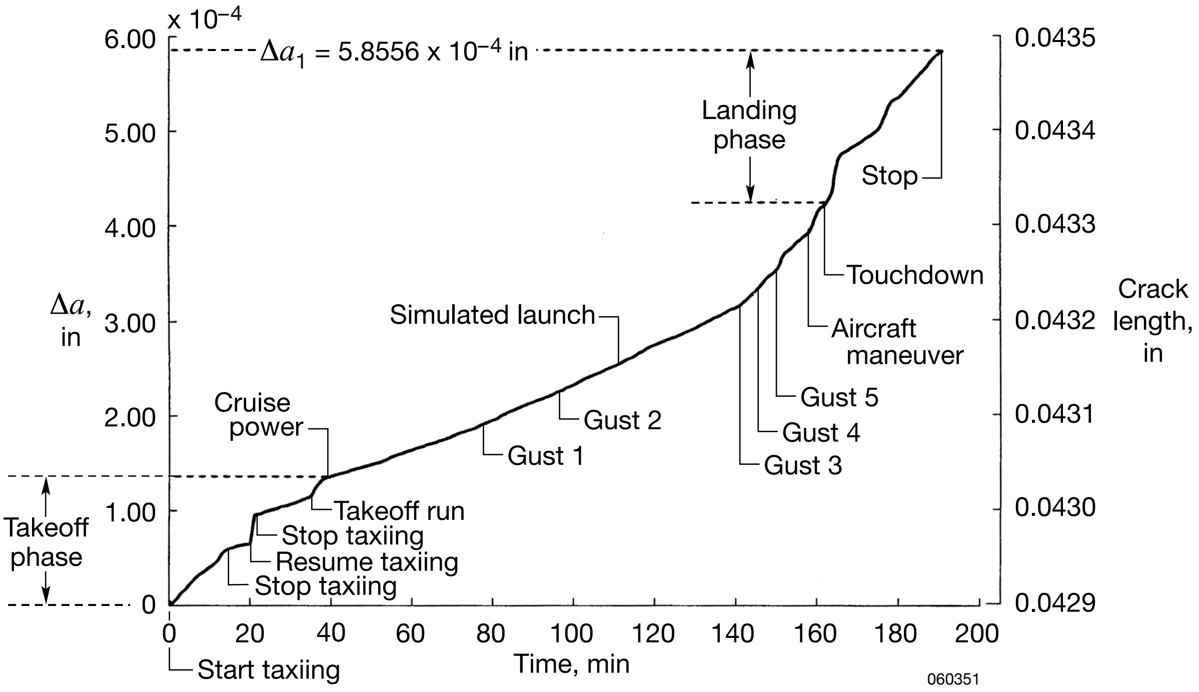


Figure 27. Crack growth curve for the B-52B rear right hook (VBR); B-52B carrying the Hyper-X launching vehicle; captive flight.

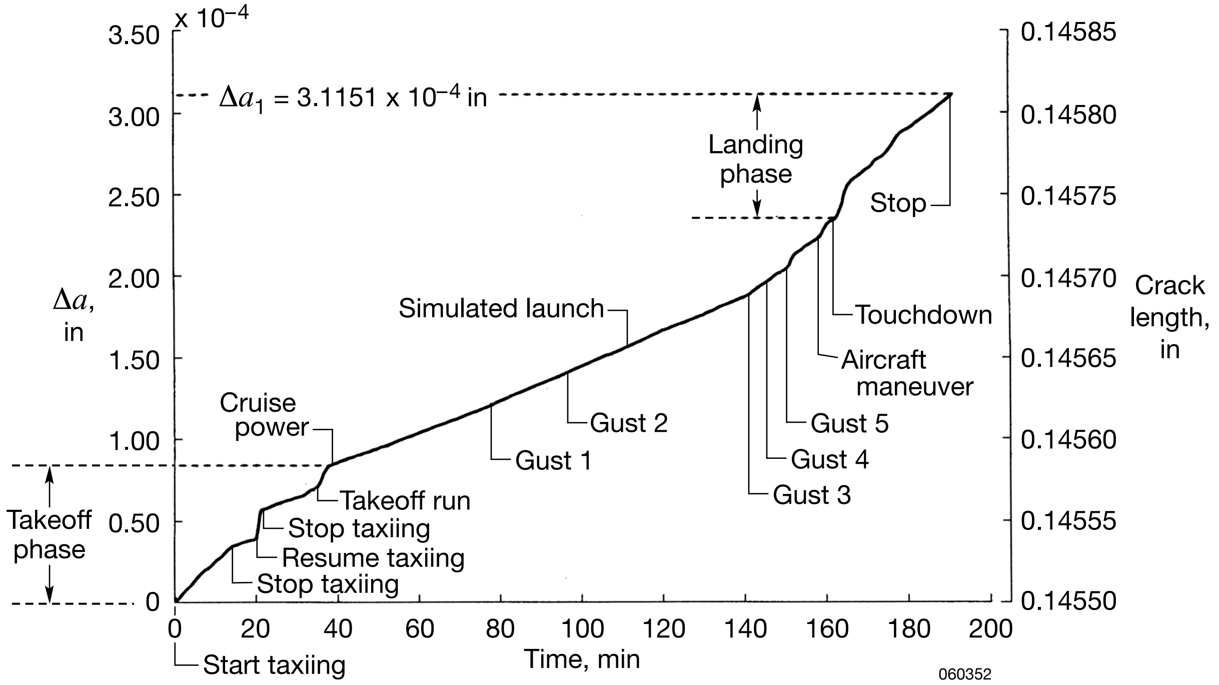


Figure 28. Crack growth curve for the Pegasus pylon front left hook (VPFL); B-52B carrying the Hyper-X launching vehicle; captive flight.

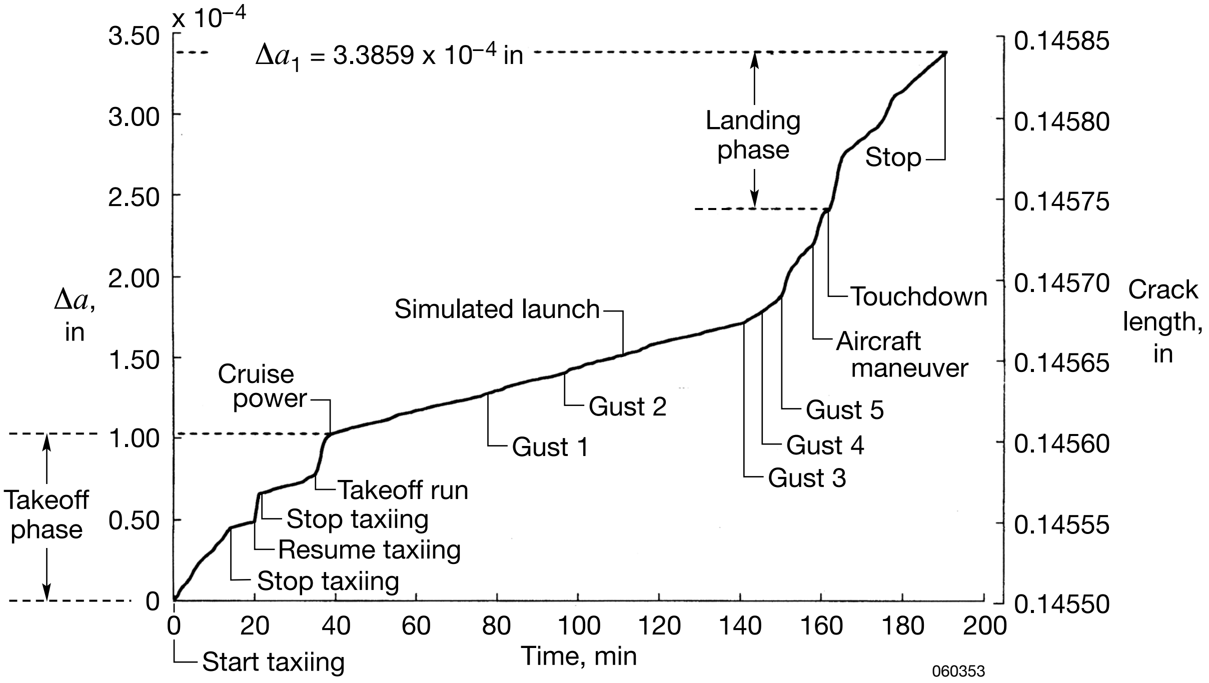


Figure 29. Crack growth curve for the Pegasus pylon front right hook (VPFR); B-52B carrying the Hyper-X launching vehicle; captive flight.

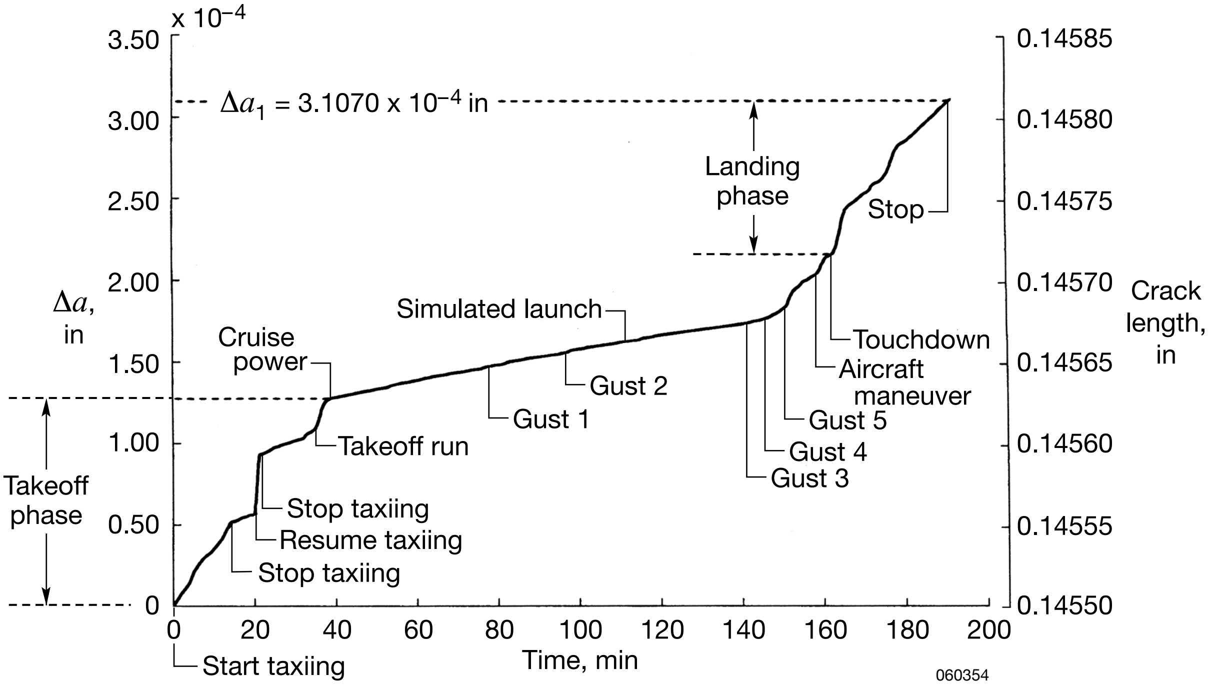


Figure 30. Crack growth curve for the Pegasus pylon rear left hook (VPRL); B-52B carrying the Hyper-X launching vehicle; captive flight.

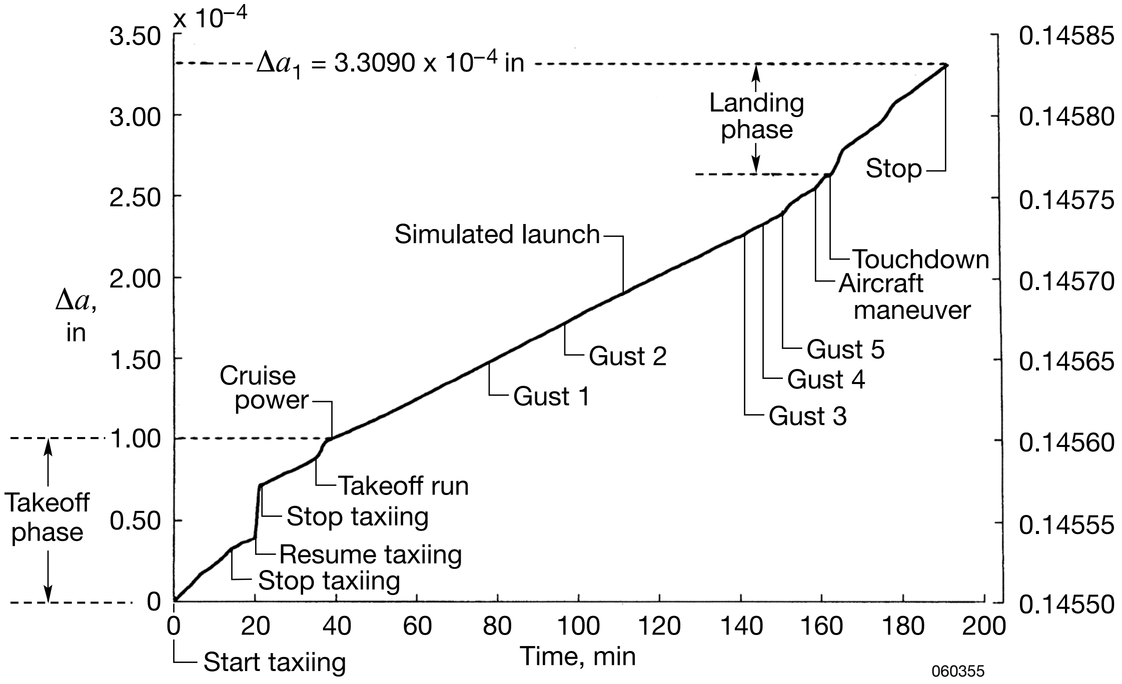


Figure 31. Crack growth curve for the Pegasus pylon rear right hook (VPRR); B-52B carrying the Hyper-X launching vehicle; captive flight.

REFERENCES

1. Ko, William L., A. L. Carter, W. W. Totton, and J. M. Ficke, *Application of Fracture Mechanics and Half-Cycle Method to the Prediction of Fatigue Life of B-52 Aircraft Pylon Components*, NASA TM-88277, 1989.

2. Ko, William L., *Prediction of Service Life of Aircraft Structural Components Using the Half-Cycle Method*, NASA TM-86812, 1987.

3. Ko, William L. and Richard Monaghan, *Practical Theories for Service Life Prediction of Critical Aerospace Structural Components*, NASA TM-4354, 1992.

4. Ko, William L., Richard C. Monaghan, and Raymond H. Jackson, *Practical Theories for Service Life Predictions of Critical Aerospace Structural Components*. Presented at the *4th International Conference on Structural Failure, Product Liability and Technical Insurance*, Vienna, Austria, July 6–9, 1992. Reprinted from Rossmanith, H. P., *Structural Failure, Product Liability and Technical Insurance IV*, Elsevier Science Publishers, Amsterdam, the Netherlands, pp. 495–504, July 1993.

5. Ko, William L., *Aging Theories for Establishing Safe Life Spans of Airborne Critical Structural Components,* NASA TP-212034, 2003.

6. Ko, William L. and Tony Chen, *Extended Aging Theories for Predictions of Safe Operational Life of Critical Airborne Structural Components*, NASA TP-2006-213676, 2006.

7. Ko, William L. and Lawrence S. Schuster, *Stress Analyses of B-52 Pylon Hooks,* NASA TM-84924, 1985.

8. Ko, William L., *Stress Analysis of B-52B and B-52H Air-Launching Systems Failure-Critical Structural Components*, NASA TP-2005-212862, 2005.

9. Ko, William L., *Stress Analysis of B-52 Pylon Hooks for Carrying the X-38 Drop Test Vehicle*, NASA/TM-97-206218, 1997.

10. Hodgeman, Charles D., *Standard Mathematical Tables*, 11th ed., Chemical Rubber Publishing Co., Cleveland, Ohio, pg. 252, 1957.

11. Barrois, W. and E. L. Ripley, *Fatigue of Aircraft Structures*, The Macmillan Co., New York, 1963.

12. Starky, W. L. and S. M. Marco, *Effects of Complex Stress-Time Cycles on the Fatigue Properties of Metals*. Transaction of the ASME, pp. 1329–1336, August 1957.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 31-01-2007	**2. REPORT TYPE** Technical Publication	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE Incorporation of Half-Cycle Theory Into Ko Aging Theory for Aerostructural Flight-Life Predictions	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Ko, William L.; Tran, Van T.; and Chen, Tony	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Dryden Flight Research Center P.O. Box 273 Edwards, California 93523-0273	**8. PERFORMING ORGANIZATION REPORT NUMBER** H-2673
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSORING/MONITOR'S ACRONYM(S)** NASA
	11. SPONSORING/MONITORING REPORT NUMBER NASA/TP-2007-214608

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified -- Unlimited
Subject Category 39 Availability: NASA CASI (301) 621-0390 Distribution: Standard

13. SUPPLEMENTARY NOTES

Ko, Tran, Chen, Dryden Flight Research Center

14. ABSTRACT

The half-cycle crack growth theory was incorporated into the Ko closed-form aging theory to improve accuracy in the predictions of operational flight life of failure-critical aerostructural components. A new crack growth computer program was written for reading the maximum and minimum loads of each half-cycle from the random loading spectra for crack growth calculations and generation of in-flight crack growth curves. The unified theories were then applied to calculate the number of flights (operational life) permitted for B-52B pylon hooks and Pegasus® adapter pylon hooks to carry the Hyper-X launching vehicle that air launches the X-43 Hyper-X research vehicle. A crack growth curve for each hook was generated for visual observation of the crack growth behavior during the entire air-launching or captive flight. It was found that taxiing and the takeoff run induced a major portion of the total crack growth per flight. The operational life theory presented can be applied to estimate the service life of any failure-critical structural components.

15. SUBJECT TERMS

Crack growth curves, Fatigue cracks, Half-cycle theory, Operational life predictions, Structural aging theory

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER ***(Include area code)***
U	U	U	UU	88	(301) 621-0390